(2)

RL-TR-91-11
In-House Report
March 1991

PROCEEDINGS OF THE FIFTH CONFERENCE ON KNOWLEDGE-BASED SOFTWARE ASSISTANT

Louis J. Hoebel and Douglas A. White, Editors

Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

RL-TR-91-11 has been reviewed and is approved for publication.

APPROVED: [illegible]

SAMUEL A. DINITTO, JR.
Chief, C2 Software Technology Division
Directorate of Command and Control

APPROVED: Raymond P. Urtz Jr

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command and Control

FOR THE COMMANDER: Ronald Raposo

RONALD RAPOSO
Directorate of Plans & Programs

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)

2. REPORT DATE
March 1991

3. REPORT TYPE AND DATES COVERED
In-House 24-28 Sep 90

4. TITLE AND SUBTITLE
PROCEEDINGS OF THE FIFTH CONFERENCE ON KNOWLEDGE-BASED SOFTWARE

5. FUNDING NUMBERS
PE - 63728F
PR - 2532
TA - PR
WU - OJ

6. AUTHOR(S)
Louis J. Hoebel, Douglas A. White

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Rome Laboratory (COES)
Griffiss AFB NY 13441-5700

8. PERFORMING ORGANIZATION REPORT NUMBER
RL-TR-91-11

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (COES)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Douglas A. White/COES/(315) 330-3564

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
This report contains the proceedings of the 5th Annual Conference on the Knowledge-Based Software Assistant sponsored by the Directorate of Command and Control at Rome Laboratory (formerly Rome Air Development Center). This conference was held at the Sheraton Convention Center in Liverpool, New York, from 24 through 28 September 1990. This annual conference provides a forum for exchanging technical and managerial views on the progress of the Rome Laboratory program for developing a knowledge-based life cycle paradigm for large software projects, the Knowledge-Based Software Assistant. The KBSA will provide intelligent assistance to system builders in producing quality mission-critical computer resources (MCCR). Software developed using the KBSA is expected to be more responsive to changing requirements, more reliable, and more revisable than software produced using current practices. The KBSA will improve software practices by providing machine-mediated support to decision makers, formalizing the processes associated with software development and project management, and providing a corporate memory for projects. The KBSA will utilize artificial intelligence, automatic programming, knowledge-based engineering, and software environment technology to achieve the goal of providing an integrated environment for developing MCCR systems.

14. SUBJECT TERMS Artificial intelligence, software development, knowledge-based systems, knowledge-based programming, intelligent assistant, automatic programming, programming environment

15. NUMBER OF PAGES
504

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	U/L

Table of Contents

Section 1: Papers

KBSA_5 Keynote Address

Physical Computation

Dr. Geoffrey C. Fox
Director
Northeast Parallel Architectures Center
at
Syracuse University

Dr. Fox will describe new computing models based on analogies with physical systems. These include and extend ideas such as simulated annealing, neural networks, cellular automata, and genetic algorithms. These new methods parallelize well and their performance scales well to large systems. Applications include modelling the turmoil in eastern Europe and many multiple robot arms in confined environments, clustering and track finding.

SECTION 1

PAPER PRESENTATIONS

An Overview of Rome Air Development Center's (RADC) Software Life Cycle Support Environment

Deborah A. Cerino
Frank S. LaMonica

Rome Air Development Center (COEE)
Griffiss Air Force Base NY 13441

Abstract

In 1983, RADC, under an exploratory development effort entitled "C3I Support Environment Definition", established the requirements for a computer-based environment of software engineering techniques and tools. That effort defined the key components of a software engineering environment which supports the complete software life cycle (requirements, design, code, test, etc). Since then, RADC has developed this environment, called the Software Life Cycle Support Environment (SLCSE), as well as planned for a number of related Research and Development (R&D) efforts to support its evolution. This paper describes the concept of SLCSE, its related efforts, and the investigation of insertion of knowledge-based technology into SLCSE.

What is SLCSE?

The SLCSE, (pronounced "slice") is a computer-based environment of integrated software tools which supports the full-scale development and post-deployment phases of the mission critical computer system (MCCS) software life cycle. SLCSE provides support for the various phases and inter-phase activities of the software life cycle including requirements specification and analysis, design, coding, unit/integration testing, quality assurance (QA), verification and validation (V&V), project management, and configuration management. It supports these activities in the context of a "conventional" yet, in many aspects, state-of-the-art software engineering environment.

As illustrated in Figure 1, SLCSE consists of three major subsystems: the user interface, database, and toolset.

The user interface is window-oriented and menu-driven, providing a common and consistent style of operation to all its users. While the user interface is consistent in style, the tools and database views a user has access to are governed by the various roles the user has been assigned by the project manager.

The database subsystem consists of an Entity-Relationship (E-R) database interface to an underlying relational database engine. The E-R database schema currently models the formal data requirements of DOD-STD-2167A, Defense System Software Development. The complete and comprehensive database can be used to generate each of the seventeen (17) formal Data Item Description (DIDs) in accordance with DOD-STD-2167A. The SLCSE schema can be replaced or modified to support other life cycle models.

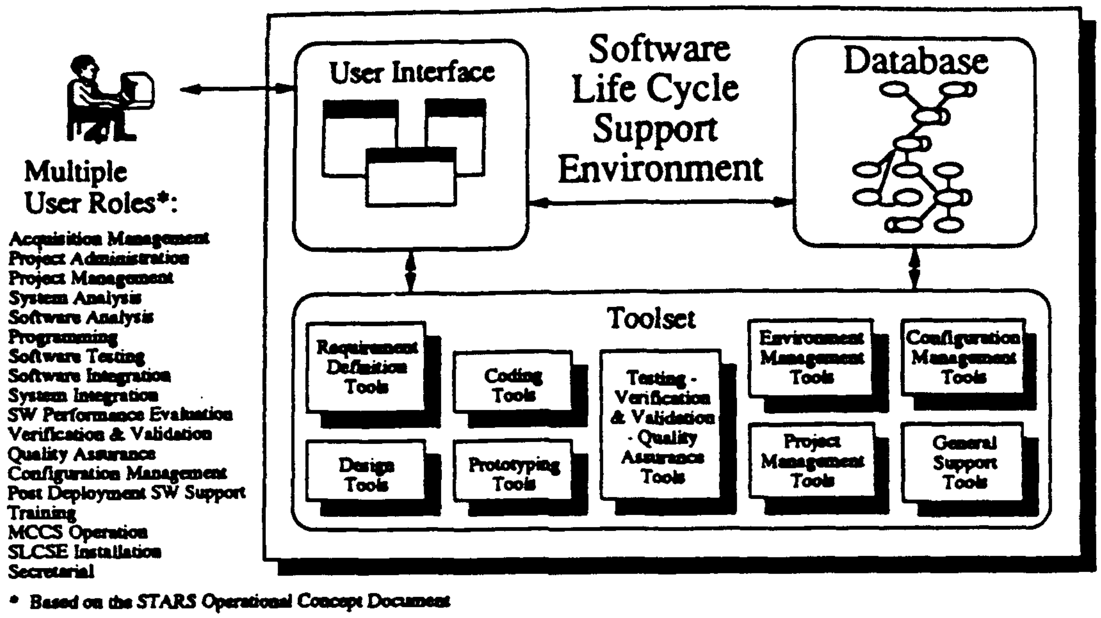


FIGURE 1: SLCSE Subsystems

The user interface and database are considered the "integrating framework" of the environment and are the two subsystems within which the tools are integrated. Tools are invoked from the SLCSE user interface and interact through the project database which stores and manages all of the formal project data created throughout the life cycle.

SLCSE has been designed using a flexible and evolvable tools-based approach where a virtually unlimited number of tools can be integrated to support various software engineering methods and overall development methodologies. This "methodology independent" tool integration concept provides for the use of off-the-shelf tools, as well as tools developed specifically for SLCSE. Thus, a project manager has maximum flexibility in the choice of tools for use on a particular software development project.

Categories of tools that have been integrated into SLCSE include:

- General Support Tools
- Requirements Definition Tools
- Design Tools

- Coding Tools
- Prototyping Tools
- Quality Assurance/V&V/Testing Tools
- Configuration Management Tools
- Project Management Tools
- Environment Management Tools

The current toolset was intended to demonstrate proof-of-concept for the framework and the integration of COTS tools. As a result, it provides a representative, but by no means exhaustive, collection of capabilities supporting the full spectrum of software life cycle activities. The toolset is neither complete nor sufficient to support each of the SLCSE defined user roles (e.g., there is no acquisition management tool). The toolset is meant to evolve over time, with the integrating framework supporting and encouraging the integration of both custom fabricated and commercial-off-the-shelf (COTS) tools to meet future software project needs.

Database Architecture (Life Cycle Project Database)

The database is SLCSE's most critical and important component. It serves not only as a repository for formal life cycle information as required by DOD-STD-2167A, but also as an integrating mechanism for tools by allowing them to share information. At its highest level of abstraction, the SLCSE database appears as a single Entity-Relationship (E-R) model which conforms to the formal data requirements of DOD-STD-2167A and its associated data item descriptions. It supports the necessary persistent database objects and persistent relationships among those objects.

For purposes of managing the model's development and also providing the ultimate capability of user role oriented views of the database, the model was logically partitioned into the nine (9) subschemas (or submodels) as indicated in Figure 2.

Life cycle phase oriented support is provided by the System Requirements, Software Requirements, Design, Test and Environment subschemas. Inter-phase support is provided by the Contract, Product Evaluation, Project Management, and Configuration Management subschemas.

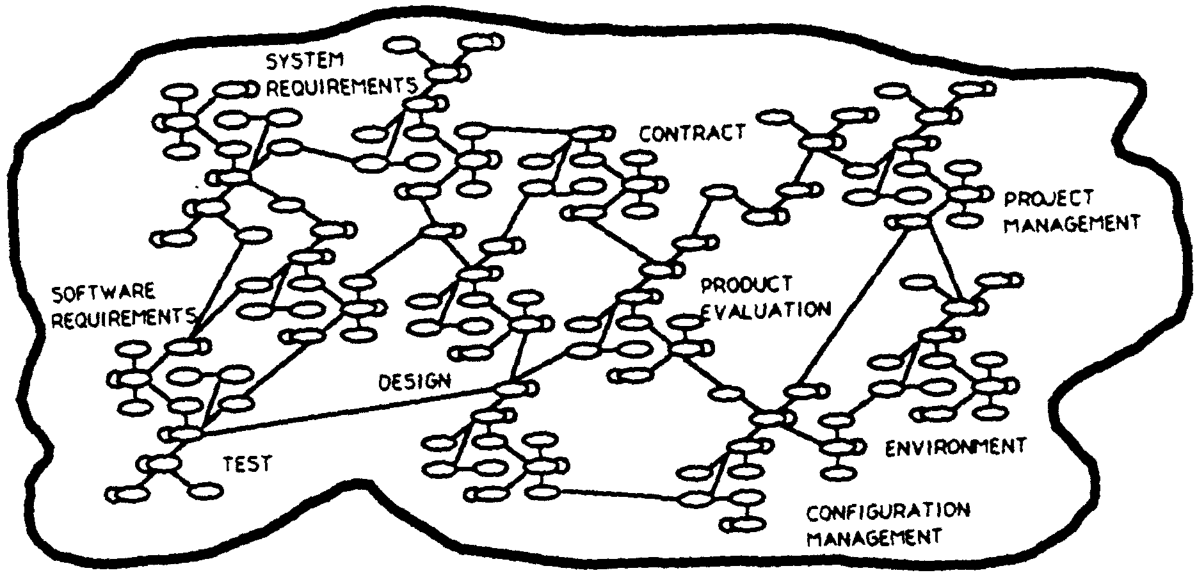


FIGURE 2: SLCSE Subschemas

Rule Base

SLCSE contains a simple rule base with a limited set of rules. A SLCSE administrator, with input from the project manager, can set up rules to be adhered to on a project. Rules follow an Ada-like syntax which allows mathematical expressions to be used as well as predefined functions. Rules can be associated with a specific tool and can be checked prior to a tool's invocation (i.e., pre-invocation) or after a tool's execution (i.e., post-execution). [1] For example, if a project manager wanted to be sure that a particular test tool was used before baselining the product, a rule could be set-up to inform the user, before using the baseliner tool, that the test tool had not been executed. This rule would only inform the user, it does not control subsequent development processes. The rule base is an initial capability; as true knowledge-based technology is added to SLCSE this rule base will be enhanced in complexity and capability.

SLCSE Platform

SLCSE operates on Digital Equipment Corporation (DEC) VAX/VMS hardware/software platforms, ranging from a MicroVAX Workstation (SLCSE was developed using a MicroVAX), to a top of the line VAX/VMS 9000. There are two database implementations currently available. Both implementations

[1]Strelich, T., SLCSE Final Technical Report, General Research Corporation, RADC-TR-89-385, February 1990.

use the SMARTSTAR* 4th generation language relational database management system. Underlying SMARTSTAR can be either DEC's Rdb/VAX relational database management system, or ShareBase Corporation's high performance database machine, (i.e., ShareBase Server) Either implementation can be used, as shown in Figure 3. SMARTSTAR was selected as the database management system (in 1986) because it was the only system which supported the necessary SQL interface to the ShareBase Server.

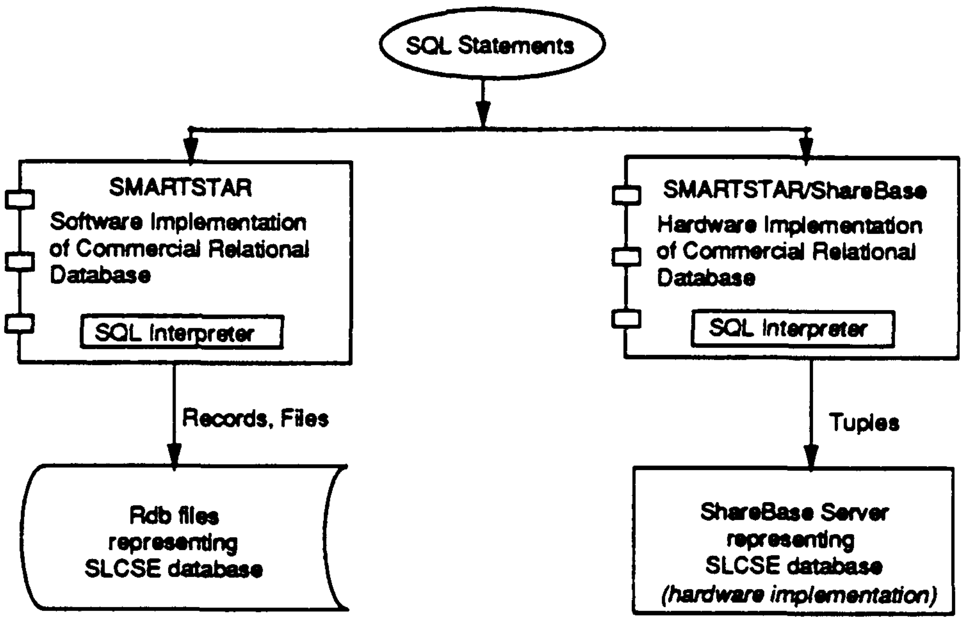

SQL Statements
SMARTSTAR
Software Implementation of Commercial Relational Database
SQL Interpreter
SMARTSTAR/ShareBase
Hardware Implementation of Commercial Relational Database
SQL Interpreter
Records, Files
Tuples
Rdb files representing SLCSE database
ShareBase Server representing SLCSE database (hardware implementation)

FIGURE 3: Database Implementations

Significant Capabilities

SLCSE contains a number of capabilities which provide a significant payoff over the course of the life cycle. As data is generated and stored by users and tools, it becomes available through database retrieval functions for use by other users and tools in subsequent activities and life cycle phases. The SLCSE integrating framework, and in particular the interaction between tools and the project database, is what makes possible many life cycle oriented technological opportunities and potential productivity gains which include the following:

• Effective Transition of Information from One Life Cycle Phase to the Next -- Because all life cycle information is available and managed on-line, SLCSE users can easily access needed information produced in previous life cycle phases (e.g., programmer accessing design information).

* SMARTSTAR is a licensed product of Signal Technology Inc., Goleta CA.

• Effective Sharing of Information by Tools -- Tool integration in SLCSE is based on the concept that tools share information; in SLCSE the life cycle project database is the medium through which tools share information.

• Life Cycle Data Traceability and Change Impact Analyses -- A workstation-based life cycle impact analysis tool (ALICIA) has been integrated into SLCSE to provide traceability as well as analysis and assessment of impacts to the developed software's requirements, design, code, test-set, etc. ALICIA enables a user to navigate the E-R model in the project database and to identify entities and relationships which are impacted by a particular change. A variety of algorithms (heuristic and deterministic) automatically identify potential impacts throughout the database.

• Automated Document Generation -- The SLCSE database supports the automated generation of DOD-STD-2167A compliant data items. All information pertaining to *the seventeen (17) formal data items (Software Requirements* Specification, Software Design Document, etc.) is stored within the entities and attributes of the SLCSE data base. Each data item is guaranteed, by design, to be compliant with the Data Item Descriptions (DIDs) of DOD-STD-2167A. Each data item is consistent with the developing software product since it is developed at the same time as the product and much of the data in the database can be populated automatically via use of tools within the SLCSE toolset.

• Software Quality Data Collection and Assessment -- The SLCSE contains a software QUality Evaluation System (QUES) that collects, analyzes, and reports on software quality at all phases of the software life cycle. Data is collected both automatically and manually. During the coding life cycle phase, 80% of the software quality data for Ada and FORTRAN code is automatically collected. The QUES reports on 13 high level software quality concerns, called factors. These include concerns such as reliability, portability, maintainability, expandability, flexibility, etc.

• Baselining and Version Control -- The SLCSE supports the software baselines in accordance with DOD-STD-2167A and includes a Baseliner Tool which will baseline (i.e., lock and create a new copy of) the entities, attributes, and relationships related to the allocated, functional, and product baselines, as well as user defined *baselines and various development configurations.* As required by DOD-STD-2167A, baselining is document driven.

SLCSE Related R & D

The RADC Software Engineering Program includes a series of R&D efforts which will either evolve SLCSE as a product or take advantage of SLCSE concepts in the development of more advanced system engineering environment technology. Of particular importance here, however, are efforts associated with knowledge-based concepts and the integration of knowledge-based tools.

It should be noted that an enhanced SLCSE which incorporates knowledge-based technology is not meant to compete with the Knowledge Based Software Assistant (KBSA) development program. The objective is to use technology spin-offs of the KBSA program which are determined mature enough to be incorporated and applied in the near term.

SLCSE Evolution

Following is a brief description of the efforts associated with evolving SLCSE as a product. These efforts, although advancing the state-of-the-art in software engineering tools, do not take advantage of knowledge-based technology:

SLCSE Project Management System (SPMS) - The objective of the SPMS is to support Government and contractor project managers in planning, tracking, and assessing developing software. The SPMS provides the capability to define tasks, milestones, configurations, schedules, costs, and responsibilities in an original project plan, and to track the progress of the actual project, identify problem areas, and return to the planning tools for project adjustment, if necessary. The SPMS provides many of these capabilities through the use of a commercial off-the-shelf tool, MacProject II, hosted on a Macintosh II Workstation.

As part of the SPMS effort, the SLCSE E-R Interface has been modified to permit access to the SLCSE database, in terms of entities and relationships, by any workstation (node) that can be configured as a DECnet node. Thus, the SPMS effort will provide SLCSE with the ability to support software development on heterogeneous computer configurations, where the actual software development activities may be performed on workstations and then (upon user direction) the results are inserted into the SLCSE project database after the workstation session is complete via the client/server approach. The SPMS is currently under development by General Research Corporation, and will be delivered to RADC in December 1990.

Ada Test and Verification System (ATVS) - The ATVS is a software test tool, in the SLCSE toolset, that provides execution coverage information regarding the amount (percentage) of Ada code that has been tested. The objective of the ATVS is to increase the user's overall confidence in the program under test; it is not to formally prove a program correct/incorrect. ATVS records the program branches (i.e., decision points and statements between the decision points) that have been executed with a particular set of test data and keeps a cumulative history of test data and execution coverage. The ideal goal is to obtain 100% execution coverage.

The ATVS supports the complete Ada language (MIL-STD-1815A). In fact, it contains the "front-end" of an Ada compiler. As part of ATVS's normal activities, the ATVS stores the number of: lines of code, branches, procedures, package bodies, specifications, enumerated literals, etc. that are contained within the source code read into ATVS. The counts of these, and many more, items are stored not only in the local ATVS database, as "raw metrics", but in the SLCSE life cycle database, in the metrics subschema. These items are then available for use by software quality analysis tools, within the SLCSE toolset. RADC's goal is to reduce the data collection burden on the current "quality" tools and allow quality tools to be concerned only with data reduction, analysis, and reporting. Other mechanisms in the environment will perform the software quality data collection (see SLCSE Measurement Instrumentation, below).

The ATVS effort was completed in July 1990. The contractor, General Research Corporation, has commercialized the ATVS, making available a more robust, fully supported, and maintained product.

SLCSE Measurement Instrumentation - To meet the above goal, of the environment performing all the data collection, RADC is planning (September 1991 RFP) to "instrument" the SLCSE (user interface and toolset). This effort will develop user transparent (as well as manual) data collection mechanisms within SLCSE such that the necessary life cycle data is collected in a complete and consistent manner to support the measurement and assessment of software quality. As a result, software quality data collection will be part of the normal activities of software development and will no longer be a separate activity. For example, in the process of performing software requirements, the requirements tools will be responsible for collecting data important to determining software quality in addition to performing their normal activities. Similarly, all SLCSE tools will collect quality data in their particular life cycle activity and store this data within the SLCSE life cycle database. This has already been accomplished for the testing life cycle phase via the ATVS. This effort will reduce the labor intensiveness of the current manual software quality data collection process.

System Engineering Support

The following effort is concerned with the development of advanced system engineering environment technology. It may take advantage of knowledge-based technology to meet its overall goals:

System Engineering Concept Demonstration - This is an exploratory development effort, initiated in September 1989, whose objective is to demonstrate the concept of an advanced computer-based environment of integrated software (CASE/CAD/CAM) tools and methods which support the Air Force computer-based system (i.e., software, firmware, and hardware) life cycle - short of actual hardware fabrication. In addition to demonstrations of advanced system engineering concepts and enabling technologies, this effort will also develop and document the system requirements, system design, and software requirements of the computer-based environment. Resulting documentation will be used for follow-on advanced development purposes.

While this effort will incorporate several SLCSE concepts, its design will capitalize on such advanced enabling technologies as heterogeneous workstation networks, object-oriented databases, reusability libraries, rule bases, knowledge-based/AI systems, and hypermedia interfaces. This effort will be completed in September 1991.

Insertion of Knowledge-Based Technology into SLCSE

As a conventional life cycle environment, SLCSE does not provide support in the area of how to perform the various software development processes. It provides the toolset, interface, and database, but it is up to the software project manager to decide which tools are to be used, and when to use them in the software development process. As a conventional software engineering environment, SLCSE does not provide intelligent support to help users (software developers) decide what to do next or how to do it. SLCSE performs a specific function or set of functions based upon user input without the knowledge or expertise of the method or process being supported. For example, SLCSE will allow a naive user to compile a FORTRAN program with the Ada compiler!

To meet this need, RADC/COEE has initiated two investigations of knowledge-based technology insertion into SLCSE, and has integrated the KBSA Project Management Assistant (PMA) into SLCSE. Following is a description of these efforts.

1) "PMA integration into SLCSE." The PMA, developed under the KBSA program (first-iteration effort), was completed in August 1990. One task of this effort was to integrate the PMA into the SLCSE. Unlike SLCSE's conventional project management tool (i.e., SPMS), the PMA formalizes software development products (e.g., components, tasks, milestones, requirements, specifications, test cases, etc) from a project management perspective and provides a language to describe the **process** by which these products were produced.[2] Project managers, can choose initially to use the PMA, then download the information that relates to DOD-STD-2167A into the SLCSE project database and then, as the project develops, can choose to use SPMS (with some of the database already populated by the PMA), or can continue to use the PMA. SLCSE provides an integrated framework of tools, it is up to the project manager to decide which tools are best suited for each particular software development effort. The PMA was developed by Kestrel Institute.

2) "Integration of Knowledge-Based and Conventional Tools." This effort, (performed by IIT Research Institute, Honeywell Systems & Research Center, and Software Productivity Solutions, Inc.) investigated the SLCSE toolset to determine which tools have best potential payoff for knowledge-based insertion. It was the conclusion of this effort that maximum payoff would result from knowledge-based insertion into the framework itself and not within the individual life cycle phase oriented tools which already exist within SLCSE. This effort was completed in June 1990.

3) "SLCSE Knowledge-Based Enhancements." This effort, recently initiated, (performed by Honeywell Systems & Research Center) is investigating the SLCSE framework to determine what changes need to be

[2]Elefante, D., "An Overview of RADC's Knowledge Based Software Assistant Program," RADC, 4th Annual KBSA Conference, Syracuse NY, 12-14 Sep 89.

made to SLCSE to support the co-existence of conventional and knowledge-based data and to determine which areas of the framework have the greatest potential payoff for knowledge-based insertion. Specific areas of knowledge-based support being investigated under this effort are:

• Support for various software development processes/methodologies within the DOD-STD-2167A life cycle model. Knowledge about the software development process and mission requirements can help determine which tools in the environment toolset should be used and at what time in the development process they should be used. A knowledge-base to contain this type of information is under investigation. In support of a process, the environment can take an active or passive role in the software development process. It can initiate actions, on behalf of the user, based on the state of the system. Or it can simply monitor the activities occurring and notify appropriate users regarding the state of the system, but not initiating or controlling any subsequent action on behalf of the user. In either an active or passive role, both cases require knowledge about a process. Support for both an active and passive environment and for knowledge within the environment to determine the appropriate times for activities to occur in the software development process, are being addressed.

• *Formulation of optimum* tool functionality for the problem domain being addressed. Knowledge about the type of application to be developed (i.e., avionics, C3I, space related) size, complexity, criticality, and project duration may help determine which tools in the SLCSE toolset are best suited for a particular software development. Since the SLCSE toolset is comprehensive, a knowledge-base to help tailor the toolset to the application is being addressed. This will result in efficient use of computing resources and computer system storage on a project by project basis.

• Management of and support for configuring the environment (hardware/software). There are numerous decisions involved in the set-up (i.e., instantiation) and configuration of SLCSE. An *effective* and *efficient* environment must contain not only the appropriate set of software development tools to meet the project's needs and support the underlying development methodology, but help in setting up users and their roles in the software process and determining what they can access in the SLCSE database. It is also necessary to efficiently configure the hardware environment for SLCSE users according to the software application. With the benefit of knowledge of similar projects and associated requirements, determining an efficient configuration for each instantiation will be a much less difficult and time consuming process. Knowledge-based support for these areas is being addressed.

• Support for database/toolset evolution. As SLCSE evolves the toolset will change, tools will be added, others modified, the user interface will be expanded, and the existing life cycle database schemas may be replaced (e.g., DOD-STD-2167A replaced with DOD-STD-2167B). Stored knowledge about the SLCSE architecture will enable automated support for accommodating these changes and will allow the environment to effectively and efficiently evolve to handle these changes.

• Support for the development of knowledge-based software. SLCSE is also being investigated to determine how it can be enhanced to support the development of knowledge-based software and if commercial knowledge-based systems (e.g. KEE, ART) can be efficiently and effectively integrated in SLCSE to help support the development of knowledge-based software. SLCSE currently supports the development of software written in the Air Force Standard Languages (i.e., Ada, FORTRAN, Jovial J73, and COBOL).

To provide support in the above areas, this effort is also investigating the changes that must occur within SLCSE to provide for the co-existence of knowledge-based and conventional data. SLCSE's E-R database was not designed to contain the knowledge-based data that will result from the insertion of knowledge-based tools/techniques. This knowledge-based data must co-exist with the conventional data, although it is not required that SLCSE remain tied to the E-R database. Potential approaches to solving this problem with respect to the near-term, mid-term, and far-term (5 yrs away) are being investigated.

Upon completion of the SLCSE Knowledge-Based Enhancement effort in September 1991, a number of technical reports will be produced documenting the results of the investigations. A Software Requirements Specification (SRS) will be developed that will contain what has been determined to be the most viable, implementable, knowledge-based techniques. The SRS will be used as input for the SLCSE Knowledge-Based Enhancement Implementation follow-on effort scheduled to commence in April 1992. The follow-on will use the SRS as a starting point, develop the design, and implement a new "Knowledge-Based SLCSE."

Natural Language Technology Insertion

Under the Small Business Innovation Research (SBIR) program, SLCSE will be investigated to determine the potential application of state-of-the-art automated text generation (natural language) technology. Of significant concern within SLCSE is that of displaying to the user, in an easy to understand format, the output of the information generated by the software development activities and supporting development tools. In the area of formal documentation, this effort will be investigating providing automated text generation from an audit of the environment's database to significantly increase the overall productivity of a user and essentially guarantee the correctness of the resulting documentation. In addition, there are numerous potential applications in the area of support for project management that are being investigated. The technical challenges are to provide this text in a meaningful, dynamic way such that the text is automatically derived from the project development activities, it does not become repetitive or awkward, and the manager can immediately obtain the reasons why certain conclusions were made about the project. Application areas which show the highest potential payoff will be implemented in a follow-on effort in the September 1991 time frame.

Technology Transition/Transfer

SLCSE beta test sites have been established at three Air Logistics Centers (ALCs) within the Air Force Logistics Command. These sites (Warner Robins AFB, GA; Hill AFB, UT; and McClellan AFB, CA) are using the SLCSE over a period of six months, on small software development projects and are testing out the various tools in the toolset for potential application to future ALC projects. These beta sites will be recommending additional tools/capabilities for inclusion in the SLCSE toolset specifically to support Logistics activities.

The MITRE Corporation is also currently assessing the SLCSE. They are assessing it from the perspective of how adequately SLCSE, with its initial toolset, provides support for Electronic Systems Division (ESD) Hanscom Air Force Base, System Program Offices (SPOs) and their contractors. As a result of this effort, MITRE will develop a "Productization Plan" for SLCSE which will describe the transition of the SLCSE from its current state, a 6.3A advanced development program to a 6.4 engineering development program. Under this effort SLCSE has been installed in the Command Center Evaluation Facility (CCEF) of ESD and a Technology Transfer Plan for transfer of the SLCSE from the CCEF to ESD SPOs and their contractors is being developed. MITRE is also evaluating emerging technologies (including knowledge-based technology) for possible future enhancements in these areas. Results of this effort, in the form of a Final Technical Report, will be available in October 1990.

Conclusion

SLCSE's goal is to aid in the software development process by providing automated tools to help in software development. It is not sufficient today to simply provide automated aids. Many times there are so many options/choices for the operation of software engineering tools that it is confusing to know where to start, which tools to use, when to use them, and how to use them. Many of the tools necessary to support the software development process are available within SLCSE, they are state-of-the-art and many times fairly complex and sophisticated to use. Knowledge-based support in the areas outlined in this paper will help SLCSE users with some of the complex decisions that they currently have to make "on their own."

Within the near term, intelligent software development support will become available in SLCSE demonstrating that a state-of-the-art software engineering environment can apply the best, most practical knowledge-based techniques to support software developers in solving real world C3I problems. As a result, the two previously diverse technology areas, software engineering and knowledge-based technology, will merge. Each technology will contribute to reducing the life cycle cost of software development and increasing software quality. Together they will make a most significant contribution that neither one alone could, realistically, hope to achieve.

AUTOMATING THE DEVELOPMENT OF SOFTWARE

Douglas R. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304
smith@kestrel.edu

Abstract: The Kestrel Interactive Development System (KIDS) provides knowledge-based support for the derivation of correct and efficient programs from formal specifications. We trace the use of KIDS in deriving an algorithm for solving a problem arising from the design of radar and sonar signals. This derivation illustrates algorithm design, a generalized form of deductive inference, program simplification, finite differencing optimizations, partial evaluation, case analysis, and data type refinement. All of the KIDS operations are automatic except the algorithm design tactics which require some interaction at present. Dozens of programs have been derived using the KIDS environment and we believe that it can be developed to the point that it can be used for routine programming.

1. INTRODUCTION

The construction of a computer program is based on several kinds of knowledge: knowledge about the particular problem being solved, general knowledge about the application domain, programming knowledge peculiar to the domain, and general programming knowledge about algorithms, data structures, optimization techniques, performance analysis, etc. We report here on an ongoing effort to formalize and automate various sources of programming knowledge and to integrate them into a highly automated environment for developing formal specifications into correct and efficient programs (cf. Balzer 1983). The system, called KIDS (Kestrel Interactive Development System), provides tools for performing deductive inference, algorithm design, expression simplification, finite differencing, partial evaluation, data type refinement, and other transformations. The KIDS tools serve to raise the level of language from which the programmer can obtain correct and efficient executable code through the use of automated tools.

A user of KIDS develops a formal specification into a program by interactively applying a sequence of high-level transformations. During development, the user views a partially implemented specification annotated with input assumptions, invariants, and output conditions (a snapshot of a typical screen appears in the Appendix). A mouse is used to select a transformation from a command menu and to apply it to a subexpression of the specification. In effect, the user makes high-level design decisions and the system carries them out.

The unique features of KIDS include its algorithm design tactics and its use of a deductive inference component. Its other operations, such as simplification and finite differencing, are well-known, but have not been integrated before in one system. All of the KIDS transformations are correctness-preserving, fully automatic (except the algorithm design tactics which currently require some interaction) and perform significant, meaningful steps from the user's point of view. Dozens of programs have been derived using the system and we believe that KIDS could be developed to the point that it becomes economical to use for routine programming.

After general discussion of the KIDS system, we summarize the derivation of a program for enumerating all solutions to the Costas array problem, which is used to generate optimal sonar and radar signals. The steps are as follows. First we build up a domain theory in order to state and reason about the problem. Then, a well-structured but inefficient backtrack algorithm (Smith 1987) is created that works by extending partial solutions. To improve efficiency we apply simplification and partial evaluation (Bjorner 1988) operations. We also perform finite differencing (Paige 1982) which results in the introduction of data structures. Next, high-level-datatypes such as sets and sequences are refined into more machine-oriented types such as bit-vectors and linked lists. Finally, the resulting code is compiled. A detailed presentation of this derivation is given in (Smith 1991).

2. USAGE OF KIDS

We present an overview of general characteristics of the KIDS system and how it is used. Currently, KIDS runs on Symbolics and SUN-4 workstations. It is built on top of Refine[1], a commercial knowledge-based programming environment (Abraido-Fandino 1987). The Refine environment provides

- an object-attribute-style database that is used to represent software-related objects via annotated abstract syntax trees;
- grammar-based parser/unparsers that translate between text and abstract syntax;
- a very-high-level language (also called Refine) and compiler. The language supports first-order logic, set-theoretic data types and operations, transformation and pattern constructs that support the creation of rules. The compiler generates CommonLisp code.

The KIDS system is almost entirely written in Refine and all of its operations work on the annotated abstract syntax tree representation of specifications in the Refine database. KIDS uses an extension of the Refine language for specifications and programs.

KIDS is a program transformation system - one applies a sequence of consistency-preserving transformations to an initial specification and achieves a correct and hopefully efficient program. The system emphasizes the application of complex high-level transformations that perform significant and meaningful actions. From the user's point of view the system allows the user to make high-level design decisions like, "design a divide-and-conquer algorithm for that specification" or "simplify that expression in context". We hope that decisions at this level will be both intuitive to the user and be high-level enough that useful programs can be derived within a reasonable number of steps.

[1] Refine is a trademark of Reasoning Systems, Inc., Palo Alto, California.

The user typically goes through the following steps in using KIDS for program development.

1. *Develop a domain theory* - The user builds up a domain theory by defining appropriate types and functions. The user also provides laws that allow high-level reasoning about the defined functions. Our experience has been that distributive and monotonicity laws provide most of the laws that are needed to support design and optimization. Recently we have added a theory development component to KIDS that supports the automated derivation of distributive laws.

2. *Create a specification* - The user enters a specification stated in terms of the underlying domain theory.

3. *Apply a design tactic* - The user selects an algorithm design tactic from a menu and applies it to a specification. Currently KIDS has tactics for simple problem reduction (reducing a specification to a library routine) (Smith 1985), divide-and-conquer (Smith 1985), global search (binary search, backtrack, branch-and-bound) (Smith 1987), and local search (hillclimbing) (Lowry 1987).

4. *Apply optimizations* - The KIDS system allows the application of optimization techniques such as simplification, partial evaluation, finite differencing, and other transformations (Blaine et al. 1988). Each of the optimization methods are fully automatic and, with the exception of simplification (which is arbitrarily hard), take only a few seconds.

5. *Apply data type refinements* - The user can select implementations for the high-level data types in the program. Data type refinement rules carry out the details of constructing the implementation.

6. *Compile* - The resulting code is compiled to executable form. In a sense, KIDS can be regarded as a front-end to a conventional compiler.

Actually, the user is free to apply any subset of the KIDS operations in any order - the above sequence is typical of our experiments in algorithm design and is followed in this paper. The screen dump in the Appendix shows the interface at the point after algorithm design when the user has just selected the Simplify operation on the command menu at the top and is pointing to an expression as the argument to the simplifier. This ability to select arguments by pointing greatly enhances the usability of a program transformation system.

3. A SESSION WITH KIDS

We have used KIDS to design and optimize algorithms for over fifty problems. Examples include optimal job scheduling, enumerating cyclic difference sets, finding graph colorings, bin packing, binary search, vertex covers of a graph, knapsack, traveling salesman tours, k-queens, linear programming (a variant of the simplex algorithm (Lowry 1987)), maximal segment sums, and sorting. On several occasions we have been able to perform new derivations before an audience. We use the Costas array problem to illustrate KIDS.

					-1
				-3	1
			-4	-1	-2
		1	-2	-4	3
	-2	3	-5	1	2
2	4	1	6	5	3

Figure 1 - Costas array of order 6 and its difference table

3.1. The Costas Array Problem

In (Costas 1984), Costas introduced a class of permutations that can be used to generate radar and sonar signals with ideal ambiguity functions. Since then there has been a flurry of work investigating various combinatorial properties of these permutations, now known as Costas arrays. No general construction has been found and the problem of enumerating Costas arrays has been explored by computer search (Silverman 1988). A *Costas array* is defined as a permutation of the set *{1 .. n}* such that there are no repeated elements in any row of its difference table (See Figure 1). The first row of the difference table gives the difference of adjacent elements of the permutation, the second row gives the difference of every second element, and so on.

3.2. Domain Theory and Specification

Before a specification can be written, the relevant concepts, operations, relationship, and properties of the problem must be defined. Thus the first, and often the hardest, step in deriving an algorithm for solving a problem is the formalization of its domain theory. KIDS provides rudimentary support for the development of domain theories. A *theory presentation* (or simply a *theory*) comprises a set of imported theories, new type definitions, function specifications with optional operational definitions, laws (axioms and theorems), and rules of inference. The domain theory for the Costas array problem is summarized below.

A hierarchic library of theories is maintained with importation as the principal link. Currently about 30% of KIDS' domain knowledge is encapsulated in 25 domain theories. The rest of its domain knowledge is represented as an unstructured collection of definitions and rules.

Users can enter definitions of new functions or create new definitions by abstraction on existing expressions. The inference system can be used to verify common properties such as associativity, commutativity, or idempotence. More interestingly, we have used a deductive inference system, called RAINBOW II, to automatically derive theorems from definitions and axioms.

A useful heuristic in constructing a domain theory is that the laws for reasoning about the domain concepts should be simple. A related notion is that over 80% of the laws needed to support design and optimization in KIDS are distributive laws - analogous to the familiar distributive laws of arithmetic:

$$a \times (b + c) = (a \times b) + (a \times c) \qquad \text{(distribute } \times \text{ over } +\text{)}$$
$$a + (b \times c) = (a + b) \times (a + c). \qquad \text{(distribute } + \text{ over } \times\text{)}$$

Consequently, our work g methodology is to favor domain concepts that have simple distributive laws whenever possible. In addition, tools have been added to KIDS that support the derivation of distributive laws for user-designated functions.

The following function (written in the Refine language) builds a difference table for a given sequence.

```
function dt (p:seq(integer)):map(tuple(integer,integer),integer)
  = {| <i,j> → p(i) - p(i - j) |  i ∈ domain(p) & j ∈ {1 .. i - 1} |}
```

It turns out that *dt* distributes nicely over concatenation of sequences:

dt([]) = {| |}

dt(concat(p,q)) = dt(p) map-union cross-dt(p,q) map-union dt(q)

where

```
function cross-dt (p:seq(integer), q:seq(integer)) : map(tuple(integer,integer), integer)
  = {| <i,j> → q(i - n) - p(i - j) |   n = size(p)
                                      & i ∈ image(lambda(k) k + n, domain(q))
                                      & j ∈ {i - n .. i - 1} |}.
```

The *dt* function entails the need for other functions: *dtrow(d,i)* returns the *i* th row of the difference table *d*; *nodups(p)* holds iff *p* does not contain duplicate occurrences of some element. Distributive laws for these concepts are straightforward. The concept that a sequence is a permutation is expressed by the notion of a bijection. This latter concept and associated laws for reasoning about it is imported with the theory called SEQUENCES-AS-MAPS.

```
function injective (M:seq(integer), S:set(integer)):boolean
  = range(M) ⊆ S
   & ∀ (i,j)(i ∈ domain(M) & j ∈ domain(M) & i ≠ j ⇒ M(i) ≠ M(j))
```

```
function bijective (M:seq(integer), S:set(integer)):boolean
  = injective(M,S) & range(M)=S
```

That is, a sequence *M* is injective into a set *S* if all elements of *M* are in *S* and no element of *M* occurs twice. A sequence *M* is bijective into a set *S* if it is injective and each element of *S* occurs in *M*. The complete Costas array theory used by KIDS is given in (Smith 1991).

We can now formulate a specification for the Costas array problem:

```
function COSTAS (n:integer )
  where $1 \leq n$
  returns $\{ p \mid bijective(p, \{1 .. n\})$
            $\& \forall(j)(j \in domain(p) \Rightarrow nodups(dtrow(dt(p),j))) \}$
```

The "where" clause states conditions assumes to hold for inputs and the "returns" clause states the output conditions - here that we want *COSTAS(n)* to generate all Costas arrays of order *n*.

3.3. Algorithm Design

The next step is to develop a correct, high-level algorithm for enumerating Costas arrays. We select a global search tactic in order to design a backtrack algorithm. The basic idea of global search (Smith 1987) is to represent and manipulate sets of candidate solutions. The principal operations are to *extract* candidate solutions from a set and to *split* a set into subsets. Derived operations include various *filters* which are used to eliminate sets containing no feasible or optimal solutions. Global search algorithms work as follows: starting from an initial set that contains all solutions to the given problem instance, the algorithm repeatedly extracts solutions, splits sets, and eliminates sets via filters until no sets remain to be split. The process is often described as a tree (or DAG) search in which a node represents a set of candidates and an arc represents the split relationship between set and subset. The filters serve to prune off branches of the tree that cannot lead to solutions.

The sets of candidate solutions are often infinite and even when finite they are rarely represented extensionally. Thus global search algorithms are based on an abstract data type of intensional representations called *space descriptors* . In addition to the extraction and splitting operations mentioned above, the type also includes a predicate *satisfies* that determines when a candidate solution is in the set denoted by a descriptor.

In addition to the above components of global search theory, there are various derived operations which may play a role in producing an efficient algorithm. Filters, described next, are crucial to the efficiency of a global search algorithm. Filters correspond to the notion of pruning branches in backtrack algorithms and to pruning via lower bounds and dominance relations in branch-and-bound. A *filter* is used to eliminate spaces from further processing. The *ideal filter* decides the question "Does there exist a feasible solution in space *s* ?". However, this is usually too expensive to compute, so instead we use a necessary condition on it. A *necessary filter* Φ can be used to eliminate spaces that do not contain solutions (Smith 1987).

The KIDS library currently contains global search theories for a number of problem domains, such as enumerating sets, sequences, maps, and integers. For the Costas array problem we select from a library a standard global search theory for enumerating sequences over a finite domain - *gs_sequences_over_finite_set.* and specialize it to the Costas array problem. This new specialized theory corresponds to the generator shown in Figure 2. This generator enumerates a superset of Costas arrays. The next design step is to derive mechanisms for pruning away such useless nodes of the search tree. The effect of this step is to incorporate additional problem-specific information into the generator in order to improve efficiency.

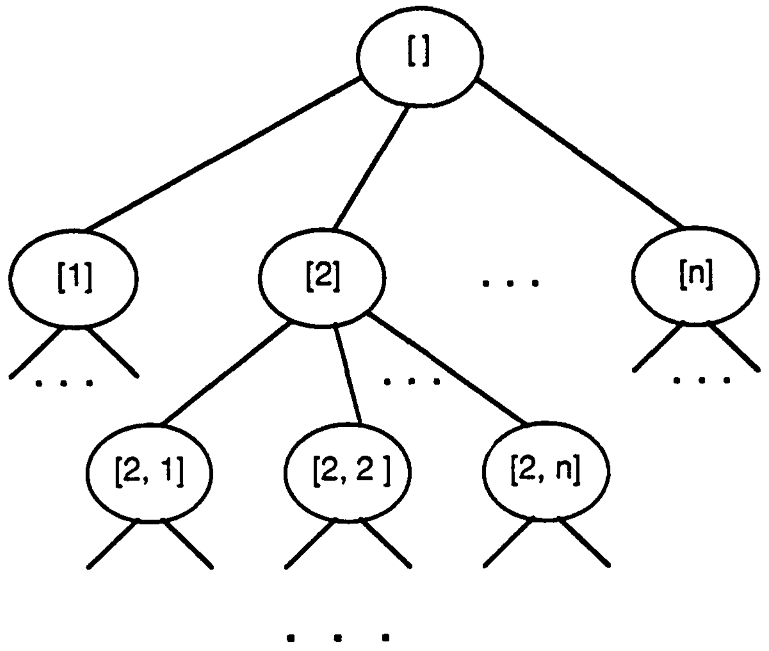


Figure 2 - Generator of sequences over the set *{1 .. n}*

KIDS invokes its inference system, called RAINBOW II, to generate a variety of necesary filters. The system presents a menu of possibilities for the user to choose from. The conjunction of any subset will result in a correct algorithm. We select

$$\forall(j)(j \in domain(V) \Rightarrow nodups(dtrow(dt(V),j))) \quad \& \quad injective(V, \{1..n\})$$

In words, the partial solution *V* must itself satisfy the constraints that there are no duplicates in the partial solution and no duplicates in any row of its difference table. It is possible to automate the selection of filters using dependency tracking but we have not done so at this writing.

Finally, the recursive Refine program in Figure 3 is produced. Note that the filter derived above is tested prior to each call to the backtracking function COSTAS-GS-AUX and thus the filter is displayed as an input invariant. Being produced as an instance of a program abstraction, this algorithm obviously has some inefficiencies. The intent of the design tactics is to produce correct, very-high-level, well-structured algorithms. Subsequent refinement and optimization is necessary in order to realize the potential of the algorithm. The interested reader should consult (Smith 1987) for the full generality of the global search model and design tactic. The tactic is sound and thus only generates correct programs.

```
function COSTAS-GS-AUX (var N: integer, var V: seq(integer))
  where 1 ≤ N & range(V) ⊆ {1 .. N}
        & INJECTIVE(V, {1 .. N})
        & ∀ (J)(J ∈ domain(V) ⇒ NODUPS(DTROW(DT(V), J)))
  returns {P | EXTENDS(P, V)
               & ∀ (J)(J ∈ domain(P) ⇒ NODUPS(DTROW(DT(P), J)))
               & BIJECTIVE(P, {1 .. N})})
  = {P | ∀ (J: integer)(J ∈ domain(P) ⇒ NODUPS(DTROW(DT(P), J)))
         & BIJECTIVE(P, {1 .. N})
         & P = V}
    ∪ reduce(∪, {COSTAS-GS-AUX(N, NEW-V) |
                   INJECTIVE(NEW-V, {1 .. N})
                   & ∀ (J)(J ∈ domain(NEW-V) ⇒ NODUPS(DTROW(DT(NEW-V), J)))
                   & ∃ (I: integer)(NEW-V = append(V, I) & I ∈ {1 .. N})})

function COSTAS (var N: integer | 1 ≤ N)
  returns {P | BIJECTIVE(P, {1 .. N})
               & ∀ (J)(J ∈ domain(P) ⇒ NODUPS(DTROW(DT(P), J)))}
  = if ∀ (J)(J ∈ domain([]) ⇒ NODUPS(DTROW(DT([]), J)))
        & INJECTIVE([], {1 .. N})
      then COSTAS-GS-AUX(N, [])
      else {}
```

Figure 3: Global search algorithm for the Costas array problem

3.4. Simplification

KIDS provides two expression simplifiers. The simplest and fastest, called the Context-Independent Simplifier (CI-SIMPLIFY), is a set of equations treated as left-to-right rewrite rules that are fired exhaustively until none apply. Some typical equations used as rewrite rules are

length([]) = 0

and

(if true then P else Q) = P.

We also treat the distributive laws in Costas array theory as rewrite rules: e.g.

injective([],S) = true

and

$$injective(append(W,a),S) = (injective(W,S) \;\&\; a \in S \;\&\; a \notin range(W)).$$

We apply CI-Simplify to the body of all newly derived programs. As a result, the conditional in program Costas-array

```
if ∀(J)(J ∈ domain([]) ⇒ nodups(dtrow(dt([]), j)))
      & injective([], {1 .. n})
    then COSTAS-GS-AUX(n, [])
   else {}
```

simplifies to *COSTAS-GS-AUX(n, [])*.

Another rule modifies a set former by replacing all occurrences of a local variable that is defined by an equality:

$$\{ C(x) \mid x=e \ \&\ P(x) \} = \{ C(e) \mid P(e) \}.$$

For example, this rule will replace *new_V* by *append(V,i)* everywhere in COSTAS-GS-AUX. This replacement in turn triggers the application of the laws for distributing *dt, dtrow, nodups,* and *injective* over *append.*

The result of applying CI-Simplify to the bodies of COSTAS and COSTAS-GS-AUX is shown in Figure 4. (For brevity we will sometimes omit or use ellipsis in place of expressions that remain unchanged after a transformation).

```
function COSTAS-GS-AUX (N: integer, V: seq(integer))
 where 1 ≤ N & range(V) ⊆ {1 .. N}
      & INJECTIVE(V, {1 .. N})
      & ∀ (J)(J ∈ domain(V) ⇒ NODUPS(DTROW(DT(V), J)))
 returns ...
 = {V | ∀ (J: integer)(J ∈ domain(V) ⇒ NODUPS(DTROW(DT(V), J)))
         & BIJECTIVE(V, {1 .. N})}
   ∪ reduce(∪, {COSTAS-GS-AUX(N, append(V, I)) |
             I ∈ {1 .. N} & I ∉ range(V)
              & INJECTIVE(V, {1 .. N})
              & ∀ (J)(J ∈ domain(V) ⇒ NODUPS(DTROW(DT(V), J)))
              & NODUPS(DTROW(DT(V), 1 + size(V)))
              & ∀ (J)(J ∈ domain(V)
                      ⇒ CROSS-NODUPS(DTROW(DT(V), J),
                                     [I - V(1 + size(V) - J) | () J ≤ size(V)]))
              & ∀ (J)(J ∈ domain(V)  ⇒ NODUPS([I - V(1 + size(V) - J) | J ≤ size(V)]))})

function COSTAS (var N: integer | 1 ≤ N)
 returns ...
 = COSTAS-GS-AUX(N, [])
```

Figure 4. Costas-array code after context-independent simplification

There are other simplification opportunities in this code. For example, notice that the predicate *injective(V, {1 .. n})* is being tested in COSTAS-GS-AUX, but it is already true because it is an input invariant. The second expression simplifier, *Context-Dependent Simplify* (CD-Simplify), is designed to simplify a given expression with respect to its context. CD-Simplify gathers all predicates that hold in the context of the expression by walking up the abstract syntax tree gathering the test of encompassing conditionals, sibling conjuncts in the condition of a set-former, etc. and ultimately the input conditions of the encompassing function. The expression is then simplified with respect to this rich assumption set

After applying CD-Simplify to the predicates of both set-formers in COSTAS-GS-AUX we obtain the following code.

```
function COSTAS-GS-AUX (N, V)
  = {V | {1 .. N} ⊆ range(V)}
    ∪ reduce(∪, {COSTAS-GS-AUX(N, append(V, I)) |
                 I ∈ {1 .. N} & I ∉ range(V)
                 & ∀(J)(J ∈ domain(V) ⇒ CROSS-NODUPS(DTROW(DT(V), J),
                                                [I - V(1 + size(V) - J)]))})
```

3.5. Partial Evaluation

Next we notice that the call to *cross_nodups* has an argument of a restricted form -- a singleton sequence. This suggests the application of partial evaluation. KIDS has the classic UNFOLD transformation that replaces a function call by its definition (with arguments replacing parameters). Partial evaluation proceeds by first UNFOLDing then simplifying.

UNFOLDing *CROSS-NODUPS(DTROW(DT(V), J),[I - V(1 + size(V) - J)])* we obtain

$$\begin{array}{l} \forall (J)\ (J \in domain(V) \\ \quad \Rightarrow \forall (I{:}\ integer, J1{:}\ integer) \\ \qquad (I \in domain(DTROW(DT(V), J))\ \&\ J1 \in domain([I - V(1 + size(V) - J)]) \\ \qquad \Rightarrow DTROW(DT(V), J)(I) \neq [I - V(1 + size(V) - J)](J1))) \end{array}$$

The following rules in the KIDS rule base

$$domain([x]) = \{1\}$$
$$x \in \{a\} = (x{=}a)$$
$$\forall (x,y)(Q(x)\ \&\ x{=}e \Rightarrow P(x)) = \forall (y)(Q(e) \Rightarrow P(e)).$$

and others are used by CI-Simplify resulting in

$$\forall (J)\ (J \in domain(V) \Rightarrow I - V(1 + size(V) - J) \notin range(DTROW(DT(V), J)))$$

KIDS produces the following code.

```
function COSTAS-GS-AUX (N, V)
  = {V | {1 .. N} ⊆ range(V)}
    ∪ reduce (∪, {COSTAS-GS-AUX(N, append(V, I)) |
          I ∈ {1 .. N} & I ∉ range(V)
          & ∀ (J) (J ∈ domain(V) ⇒ I - V(1 + size(V) - J) ∉ range(DTROW(DT(V), J)))})
```

3.6. Finite Differencing

Notice that the expression *range(V)* in Figure 8 is computed each time COSTAS-GS-AUX is invoked and that the parameter *V* changes in a regular way. This suggests that we create a new variable whose value is maintained equal to *range(V)* and which allows for incremental computation - a significant speedup. This transformation is known as finite differencing (Paige 1982). We have developed and implemented a version of finite differencing for functional programs.

Finite differencing can be decomposed into two more basic operations: abstraction followed by simplification. Abstraction of function *f(x)* with respect to expression *E(x)* adds a new parameter *c* to *f*'s parameter list (now *f(x,c)*) and adds *c=E(x)* as a new input invariant to *f*. Any call to *f*, whether a recursive call within *f* or an external call, must now be changed to supply the appropriate new argument that satisfies the invariant - *f(U)* is changed to *f(U,E(U))*. It now becomes possible to simplify various expressions within *f* and calls to *f*. In the KIDS implementation, CI-Simplify is applied to the new argument in all external calls. Within *f* we temporarily add the invariant *E(x) = c* as a rule and apply CI-Simplify to the body of *f*. This replaces all occurrences of *E(x)* by *c*. Often, distributive laws apply to *E(U(x))* yielding an expression of the form *U'(E(x))* and then *U'(c)*. The real benefit of this optimization comes from the last step, because this is where the new value of the expression *E(U(x))* is computed in terms of the old value *E(x)*.

The evolving algorithm is prepared for finite differencing by subjecting it to conditioning transformations. In this case they transform the two conjuncts

$$i \notin range(V) \;\&\; i \in \{1..n\}$$

to

$$i \in setdiff(\{1..n\}, range(V)).$$

The rationale is to group together information concerning a local variable.

We select the set difference as an expression to maintain incrementally. The changes include (1) the addition of a new input parameter, named *pool*, and its invariant to COSTAS-GS-AUX, (2) all occurrences of the term *setdiff({1..n}, range(V))* in COSTAS-GS-AUX are replaced by *pool*, (3) appropriate arguments are created and simplified for all calls to the function COSTAS-GS-AUX. The initial call to COSTAS-GS-AUX becomes

COSTAS-GS-AUX(n, [], setdiff({1..n}, range([])))

which CI-Simplifies to

COSTAS-GS-AUX(n, [], {1..n}).

The recursive call to COSTAS-GS-AUX becomes

COSTAS-GS-AUX (n, append(V, i),, setdiff({1..n}, range(append(V, i)))

which CI-Simplifies to

COSTAS-GS-AUX (n, append(V, i), pool less {i}).

Next we select

range(DTROW(DT(V), J))

and

1 + length(V)

for incremental maintenance (and naming them *dt-range* and *vsize1* respectively), KIDS produces the code in Figure 5. Notice how finite differencing introduces a meaningful data structure at this point. The concept of which elements of *{1..n}* have not yet been added to the partial solution *V* would naturally occur to many programmers who are developing a Costas array algorithm. Here it is introduced by a problem-independent transformation technique. Not only is the concept natural in the context of the problem, but its incremental computation dramatically improves the efficiency of the algorithm. Note also the need for a software database - this transformation needs global access to all invocations of a function in order to consistently modify its interface.

```
function COSTAS-GS-AUX (N,V, POOL,DT-RANGE: map(integer, set(integer)), vsize1:integer)
  where vsize1 = 1 + size(V) & DT-RANGE = {| J → DTROW(DT(V), J) | J ∈ domain(V) |}) ...
  = {V | empty(POOL)}
    ∪ reduce(∪, {COSTAS-GS-AUX(N,
                append(V, I), POOL less I,
                MAP-UNION({| vsize1 → {} |},
                          {| J → {I - V(vsize1 - J)} ∪ DT-RANGE(J)  | J ∈ domain(V) |}),
                vsize1 + 1) |
              I ∈ POOL
             & ∀(J) (J ∈ domain(V) ⇒ I - V(vsize1 - J) ∉  DT-RANGE(J))})


function COSTAS (N)
  = COSTAS-GS-AUX(N, [], {1 .. N}, { }, 1)
```

Figure 5. Costas array algorithm after finite differencing

3.7. Results and Summary

The Costas array algorithm produced by the global search tactic has been optimized and refined (a few other derivation steps such as case analysis and data type refinement are presented in (Smith 1991)). The unoptimized global search algorithm takes just under 50 minutes on a SUN-4/160 to find all 760 Costas arrays of size 9. The final optimized version finds all 760 solutions in about 5 minutes. By hand implementing the REFINE algorithm in C, using the datatype refinements we found the same 760 solutions in a second. Further incorporating an isomorph rejection mechanism further cut the time in half. We used this C version to enumerate all 18,276 Costas arrays of size 17 in about 6 days time, thus duplicating previously published results (Silverman 1988).

The derivation as presented above took place over a week's time and most of that time was spent developing the domain theory. The actual derivation and variations of it took less than a day. For the Costas array derivation, the user makes a total of 11 high-level decisions some of which involve subsidiary decisions. It would be easy to cut this number significantly by automatically applying CI-Simplify after every operation (this is not done at present). Each decision involves either selecting from a machine-generated menu, pointing to an expression, or typing a name into a text buffer. The high-level development operators encapsulate the firing of hundreds of low-level transformation rules. Excluding the time spent setting up the Costas array domain theory, the total time for the derivation is about 25 minutes on a SUN-4/160.

There are several opportunities for automating the selection and application of KIDS operations. The steps of the Costas array derivation are typical of almost all the global search algorithms that we have derived. After algorithm design,the program bodies are fully simplified, partial evaluation is applied, followed by finite differencing, and data type refinement. It is conceivable that the entire Costas array derivation could be performed automatically.

4. CONCLUDING REMARKS

The final Costas array algorithm is apparently not very complicated, however we see that it is an intricate combination of knowledge of the Costas array problem, the global search algorithm paradigm, various program optimization techniques and data structure refinement. The derivation has left us not only with an efficient, correct program but also assertions that characterize the meaning of all data structures and subprograms. These invariants together with the derivation itself serve to explain and justify the structure of the program. The explicit nature of the derivation process allows us to formally capture all design decisions and reuse them for purposes of documenting the derivation and helping to evolve the specifications and code as the user's needs change.

KIDS is unique among systems of its kind for having been used to design, optimize, and refine dozens of programs. Applications areas have included scheduling, combinatorial design, sorting and searching, computational geometry, pattern matching, routing for VLSI, and linear programming. We have had good success in using KIDS to account for the structure of many well-known algorithms. In order to demonstrate the practicality of automated knowledge-based support for software engineering, we are working toward the goal of using KIDS for its own development.

Acknowledgements: Lee Blaine, Li-Mei Gilham, Allen Goldberg, Mike Lowry, Tom Pressburger, Xiaolei Qian, and Stephen Westfold have all contributed to the KIDS system. This research was supported in part by the Rome Air Development Center under contracts F30602-86-C-0026 and F30602-88-C-0127, the Office of Naval Research under Contract N00014-87-K-0550, Personal Services Agreement No. LC90G0250F from Lockheed Corporation, and the Air Force Office of Scientific Research under Contract F49620-88-C-0033.

References

Abraido-Fandino, L. An overview of REFINE 2.0. In *Proceedings of the Second International Symposium on Knowledge Engineering*, Madrid, Spain, April 8-10, 1987.

Balzer, R., Cheatham, T.E., and Green, C. Software technology in the 1990's: using a new paradigm. IEEE Computer 16, 11 (November 1983), 39-45.

Bjorner, D., Ershov, A.P., and Jones, N.D., Eds. *Partial Evaluation and Mixed Computation*. North-Holland, Amsterdam, 1988.

Blaine, L., Goldberg, A., Pressburger, T., Qian, X., Roberts, T., and Westfold, S. Progress on the KBSA Performance Estimation Assistant. Tech. Rep.KES.U.88.11, Kestrel Institute, May 1988.

Costas, J. A study of a class of detection waveforms having nearly ideal range - doppler ambiguity properties. Proceedings of the IEEE (1984), pp.996-1009.

Goldberg, A. Reusing Software Developments. Tech. Rep., Kestrel Institute, July 1989. in *Proceedings of the KBSA Workshop*, Rome Air Development Center, Utica, NY, August 1989.

Lowry, M.R. Algorithm synthesis through problem reformulation. In *Proceedings of the 1987 National Conference on Artificial Intelligence* (Seattle, WA, July 13-17, 1987). also Technical Report KES.U.87.10, Kestrel Institute, August 1987.

Paige, R., and Koenig, S. Finite differencing of computable expressions. ACM Transactions on Programming Languages and Systems 4, 3(July 1982), 402-454.

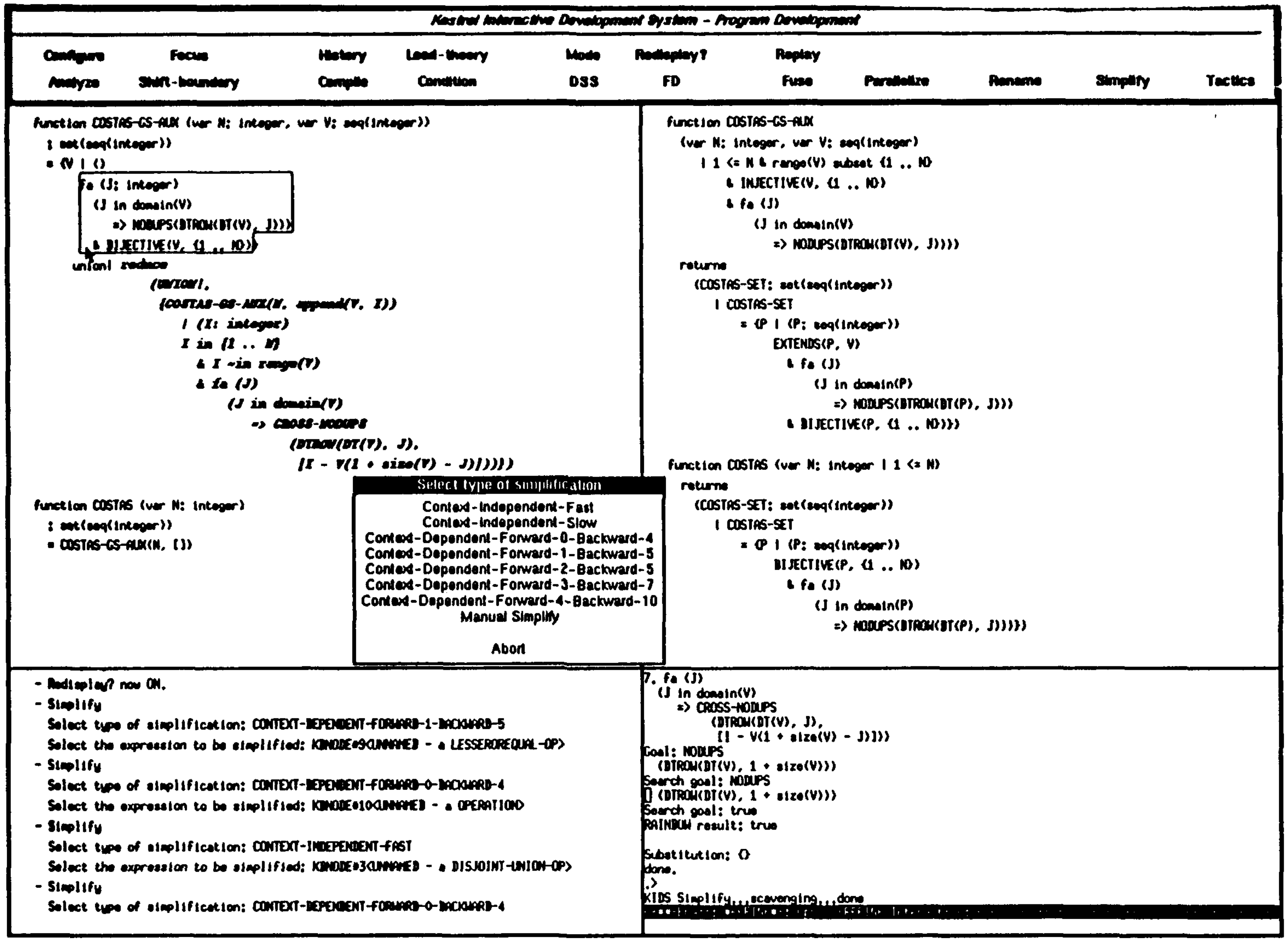
Silverman, J., Vickers, V., and Mooney, J. On the number of costas arrays as a function of array size. In *Proceedings of the IEEE* 76,7(1988), pp.851-853.

Smith, D.R. Top-down synthesis of divide-and-conquer algorithms. Artificial Intelligence 27, 1 (September 1985), 43-96. (Reprinted in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R. Waters, Eds., Los Altos, CA, Morgan Kaufmann, 1986.).

Smith, D.R. Structure and Design of Global Search Algorithms. Tech. Rep.KES.U.87.12, Kestrel Institute, November 1987. to appear in Acta Informatica.

Smith, D.R. KIDS - A Knowledge-Based Software Development System, to appear in *Automating Software Design*, M. Lowry and R. McCartney, Eds., Live Oak Press, Palo Alto, California, 1991.

Appendix


Kestrel Interactive Development System - Program Development
Configure
Focus
History
Load-theory
Mode
Redisplay?
Replay
Analyze
Shift-boundary
Compile
Condition
DSS
FD
Fuse
Parallelize
Rename
Simplify
Tactics
Select type of simplification
Context-Independent-Fast
Context-Independent-Slow
Context-Dependent-Forward-0-Backward-4
Context-Dependent-Forward-1-Backward-5
Context-Dependent-Forward-2-Backward-5
Context-Dependent-Forward-3-Backward-7
Context-Dependent-Forward-4-Backward-10
Manual Simplify
Abort

```
function COSTAS-GS-AUX (var N: integer, var V: seq(integer))
  : set(seq(integer))
  = {V | ()
      fa (J: integer)
        (J in domain(V)
          => NODUPS(DTROW(DT(V), J)))
      & BIJECTIVE(V, {1 .. N})}
     union! reduce
           (UNION!,
            {COSTAS-GS-AUX(N, append(V, I))
              | (I: integer)
              I in {1 .. N}
                & I ~in range(V)
                & fa (J)
                    (J in domain(V)
                      => CROSS-NODUPS
                           (DTROW(DT(V), J),
                            [I - V(1 + size(V) - J)]))})

function COSTAS (var N: integer)
  : set(seq(integer))
  = COSTAS-GS-AUX(N, [])
```

```
function COSTAS-GS-AUX
  (var N: integer, var V: seq(integer)
    | 1 <= N & range(V) subset {1 .. N}
        & INJECTIVE(V, {1 .. N})
        & fa (J)
            (J in domain(V)
              => NODUPS(DTROW(DT(V), J))))
  returns
    (COSTAS-SET: set(seq(integer))
      | COSTAS-SET
          = {P | (P: seq(integer))
              EXTENDS(P, V)
                & fa (J)
                    (J in domain(P)
                      => NODUPS(DTROW(DT(P), J)))
                & BIJECTIVE(P, {1 .. N})})

function COSTAS (var N: integer | 1 <= N)
  returns
    (COSTAS-SET: set(seq(integer))
      | COSTAS-SET
          = {P | (P: seq(integer))
              BIJECTIVE(P, {1 .. N})
                & fa (J)
                    (J in domain(P)
                      => NODUPS(DTROW(DT(P), J)))})
```

```
- Redisplay? now ON.
- Simplify
  Select type of simplification: CONTEXT-DEPENDENT-FORWARD-1-BACKWARD-5
  Select the expression to be simplified: KBNODE#9<UNNAMED - a LESSEROREQUAL-OP>
- Simplify
  Select type of simplification: CONTEXT-DEPENDENT-FORWARD-0-BACKWARD-4
  Select the expression to be simplified: KBNODE#10<UNNAMED - a OPERATION>
- Simplify
  Select type of simplification: CONTEXT-INDEPENDENT-FAST
  Select the expression to be simplified: KBNODE#3<UNNAMED - a DISJOINT-UNION-OP>
- Simplify
  Select type of simplification: CONTEXT-DEPENDENT-FORWARD-0-BACKWARD-4
```

```
7. fa (J)
  (J in domain(V)
    => CROSS-NODUPS
         (DTROW(DT(V), J),
          [I - V(1 + size(V) - J)]))
Goal: NODUPS
  (DTROW(DT(V), 1 + size(V)))
Search goal: NODUPS
[] (DTROW(DT(V), 1 + size(V)))
Search goal: true
RAINBOW result: true

Substitution: {}
done.
.>
KIDS Simplify...scavenging...done
```

THE ROLE OF DERIVATIONAL ANALOGY IN REUSING PROGRAM DESIGN

Sanjay Bhansali and Mehdi T. Harandi

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Avenue, Urbana, IL 61801
Telephone: (217)-244-5915

Abstract

In developing and subsequent maintenance of software systems there are numerous occasions when a problem being solved is identical or bears a similarity to a problem that has been solved earlier. By recognizing this similarity and modifying or reusing the original solution, it is possible to synthesize a program for the new problem with much less effort. We describe a system, APU, that generates shell scripts for Unix from a formal problem specification and uses derivational analogy to reuse software at both the code and design level, in order to improve the efficiency of program development.

1 INTRODUCTION

In developing and subsequent maintenance of software systems there are numerous occasions when a problem being solved is identical or bears a similarity to a problem that has been solved earlier. By recognizing this similarity and modifying or reusing the original solution, it is possible to synthesize a program for the new problem with much less effort.

One of the problems in trying to reuse or modify software is that the initial design of a system is done by a small group of people who have a thorough understanding of the problem and the domain. Typically, the documentation for the design decisions, as well as for the implemented code, is inadequate. As a result it is difficult to reuse or modify existing code and programmers often spend a large part of their time in either trying to understand code or in re-implementing it [9].

The past two decades have seen the development and maturing of the transformational paradigm for program synthesis [20]. As a result it is possible to automate the synthesis of medium-sized, non-trivial programs. An implication of this is that the steps involved in the derivation of a program can be recorded simultaneously with a program development. The derivation captures the design of a software system from an initial high-level specification to the final low-level code. This makes it possible to automatically replay the steps of the design in a new, similar context.

We have developed a knowledge-based, transformational system, APU [1] that has been used to synthesize programs from their high-level specifications [4,3]. APU makes extensive reuse of code both at the subroutine and design level. In this paper, we focus on the ability of APU to recognise problems that are analogous to a problem in its library of previously solved problems, and to use the derivation of the analogous problem to synthesize a program for the new problem.

1.1 UNIX PROGRAMMING

The domain of programming for our system is the Unix operating system environment. Unix has a rich set of commands, which can be considered similar to a library of standard subroutines. Consequently, the work expended in building this library of subroutines is saved. However, it should be pointed out that Unix programming is very similar to conventional programming, and has all the commonly used control structures of programming languages like conditionals, loops, sequences, etc. Thus the ideas presented here are just as applicable to other target domains and programming languages.

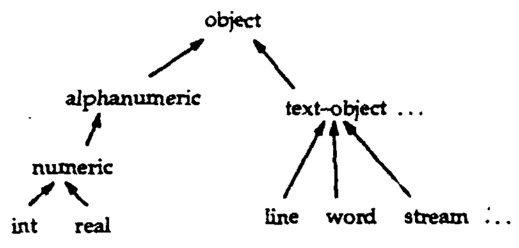
2 OVERVIEW OF APU

APU consists of two major components - a *knowledge base* and a *program generator.*

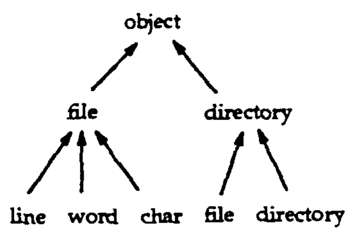
The *knowledge base* contains a description of the Unix *commands*, the set of primitive *objects, functions* and *predicates* used in problem specification, *rules* for decomposing problems and a *library* of previously solved problems. The objects and predicates are represented in abstraction hierarchies (fig. 1). Rules are similar to horn clauses used in Prolog, and are formulated in terms of the most abstract objects and predicates in the hierarchies and instantiated to particular cases for specific problems. The abstraction hierarchies and the formulation of the rules forms the basis of APU's analogical reasoning.

The *program generator* consists of a *planner* and an *analogical reasoner.* The planner uses a top-down design methodology with stepwise refinement to transform a specification

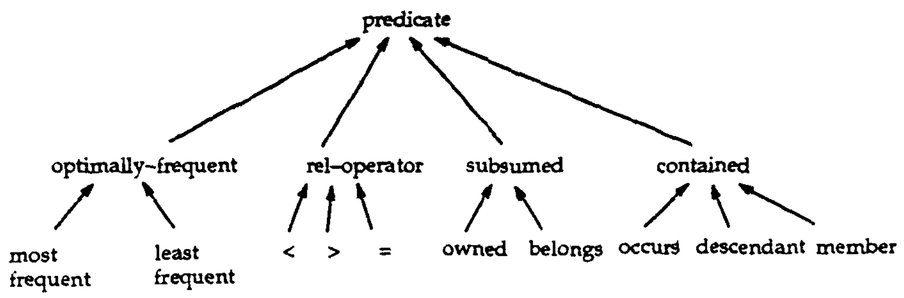
[1] *A*utomated *P*rogrammer for *U*nix



(a) ISA hierarchy for objects



(b) CONTAINED hierarchy for objects



(c) Abstraction hierarchy for predicates and functions

Figure 2. Peep-hole views of the various hierarchies in the concept dictionary.

into a program. Loosely, the methodology consists of stating a general problem, decomposing it into sub-problems, and then solving the sub-problems independently. The planner uses a backward chaining mechanism to retrieve rules whose antecedents match a given goal, and uses the rule body to decompose the goal into simpler sub-goals. This process is repeated recursively till a primitive level is reached where each sub-goal can be solved using a single Unix command or subroutine. Finally the various commands and subroutines are combined together using the *pipe* and *sequence* connectors of Unix.

Once a problem is solved by a planner, it may be stored in the plan library together with its derivation. A derivation of a problem consists of the sub-goal structure of the problem showing the decomposition of each goal into its sub-goals. With each sub-goal the following information is stored: 1) The sub-plan used to solve it. Thus the derivation has a recursive tree-like structure; 2) The rule applied to decompose the problem; 3) The set of other applicable rules; 4) The binding of variables occurring in the goal; and 5) The *type* of each variable.

When plans are stored, they are indexed using certain heuristics described in section 4.1. Besides indexing the top-level goal, we also index various intermediate-level goals.

3 AN EXAMPLE: COUNT FILES UNDER A DIRECTORY

Before describing APU's analogical reasoner, we show, very briefly, the derivation of a simple example to illustrate the program synthesis process in APU. The example is to synthesize a program to compute the number of files under a directory. A problem specification in APU is given by specifying the inputs, outputs and the pre- and post- onditions of the program. The input and output arguments are taken from a set of primitive objects recognized by the system. The pre- and post-conditions are written in a lisp like language augmented by the quantifiers *FORALL, EXISTS, SOME* and *SET* [19]. The problem for counting files in this notation is specified as follows:

```
NAME: count-files
INPUT: ?d :directory
OUTPUT: ?n :integer
PRECOND: true
POSTCOND : (= ?n (card (SET (?f :file)
                       :SUCH-THAT (occurs ?f ?d))))
```

The above specifies a program named *count-files* that takes as input a directory, *?d*, and produces an integer, *?n*, which is equal to the number of files in the directory. The specification *SET (x) :SUCH-THAT p* means "the set of all x's for which p is true", and *card* is a system-understood function that takes a set as input and returns its cardinality.

Fig 2 shows the decomposition of this problem into sub-problems. The first rule is used because there is no command in Unix to count (compute the cardinality of) the set of files under a directory. So APU uses a rule to map the set of files to a set of lines and count the set of lines. The mapping is done by listing the files, with each file on a separate line. Again there is no Unix command to list files in separate lines, so APU uses a rule that lists a set of objects (directory-objects) that include files, and selects files from the list. The set of commands used are *ls -l* (for listing directory-objects), grep "^-" (for selecting files) and *wc -l* (for counting lines). These commands are then connected together using the pipe connector to produce the final program:

```
ls -l | grep "^-" | wc -l
```

4 USING DERIVATIONAL ANALOGY

The first step in deriving a program by analogy is to recognize a problem, called the *source analogue*, which bears a significant similarity to the current problem, called the

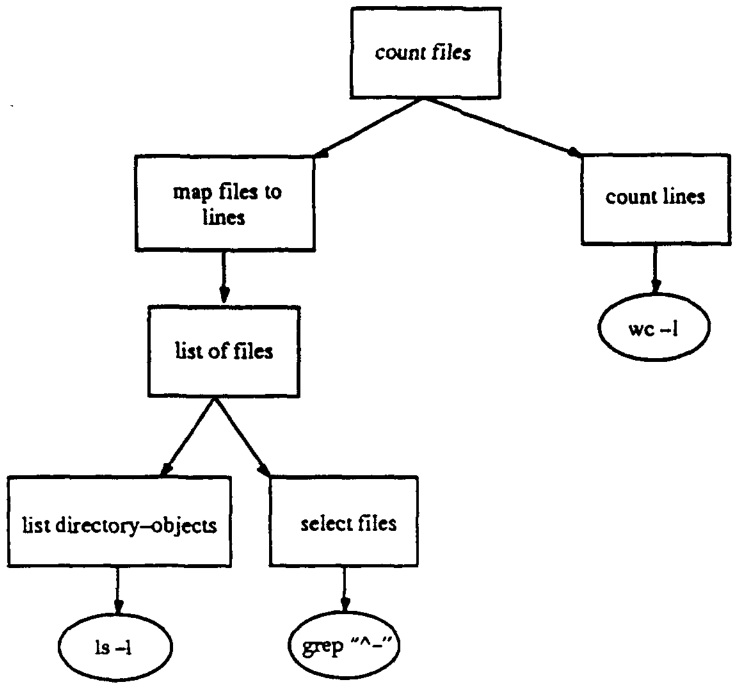


Figure 2. Derivation of program to count files under a directory

target analogue. Since our main objective is to speed up the derivation of a program, it is important for the system to be able to quickly find a source analog, if it exists.

4.1 RECOGNIZING ANALOGUES

To get an intuitive feel for the kind of knowledge required to find analogue matches, we consider an example:

P1: *Remove all files under a given directory that are bigger than 10K.*

Suppose we have already solved the following problem:

P2: *Delete all processes with cpu times greater than 1000 sec.*

An algorithm to solve this might be: *Form a list of all the processes, select all processes whose cpu time is greater than 1000 sec., retrieve their process-id's, and kill the processes.*

Seeing this solution, we can easily derive a solution for P1: *Form a list of all the files, select those whose size is greater than 10 K, retrieve their file names, and remove the files.*

Both the problems involve deleting objects and comparing numbers. But this is not the main reason that the analogy worked. To see this, consider another problem:

P3: *Change the names of all files that do not have write access by appending an extension* .read *to them.*

This problem has nothing to do with deleting objects or comparing numbers, but the same program structure can be used to solve it: *Form a list of all files, select those files that do not have write access, retrieve their names, and append the extension* .read *to them.*

The reason the analogy worked in both cases is that they are both instances of the *search-and-process* paradigm. They involve a search for a particular object among several similar objects and then some processing on a particular attribute of that object.

The success of the analogical reasoning system is, therefore, contingent on detecting the occurrence of such paradigms from the problem specification. At the same time, note that P2 seems to be "more" analogous to P1 than to P3. Given both P2 and P3 in the knowledge base, to solve P1, we would like P2 to be retrieved rather than P3. Thus, the analogue matcher should return not just a possible source analogue but the best possible source analogue in the knowledge base.

We use a set of four heuristics to index and retrieve problems in the plan library:

1. Functional Class Heuristic. One way of detecting analogies is to see whether two programs belong to the same abstract class. Program classes can be identified by considering the top-level strategy used in decomposing the problem. In our system these correspond roughly to the outermost construct used in writing specifications.

We illustrate this point with an example. Consider the post-conditions for the problems P1 and P2 given above:

```
P1: (NOT (EXIST (?f :file)
          :SUCH-THAT (and (occurs ?f ?d) (> (size f) 10))))
P2: (NOT (EXIST (?p :process)
          :SUCH-THAT (and (owned ?p ?u) (> (cpu-time ?p) 1000))))
```

The postconditions of both problems are of the form: *(NOT (EXIST (?x : ...) :SUCH-THAT (and ?constraint1 ?constraint2)))*. A construct like *(NOT (EXIST (?x :...) :SUCH-THAT ?constraints)* is suggestive of a particular strategy for solving problems: *Find all* ?x *that satisfy the given ?constraints and delete them.* Therefore the basic structure of the two solutions should be analogous.

Similarly, a program that has a postcondition of the following form: *(SET (?x_1?x_2) :SUCH-THAT ?constraints (Tuple ?x_1?x_2))* , which describes a set of tuples $<?x_1, ?x_2>$ satisfying the constraints *?constraints*, suggests a *divide-and-conquer* strategy: *First form two separate lists of all ?x_1 and all ?x_2 satisfying the independent*[2] *constraints, and then take a join of the two lists that satisfy all the constraints.*

The other quantifiers and logical connectives result in analogous strategies for writing programs. APU creates a table of such program classes and uses them to index problems.

[2]An independent constraint on a ?x_1 is a predicate that does not contain ?x_2 as an argument and vice versa

When a new problem is seen, the system computes the functional class to which it belongs and retrieves all problems stored under that class.

2. Systematicity Heuristic. This heuristic is based on the *systematicity principle* proposed by Gentner [11]: *A predicate that belongs to a mappable system of mutually interconnecting relationships is more likely to be imported into a target than an isolated predicate.*

We adopt a modified form of the systematicity principle and state that: *if the input and output arguments of two problem specifications are instances of the same system of abstract relationship, then the two problems are more likely to be analogous.*

To implement this heuristic, APU looks at each of the conjunctive constraints in the postconditions of a problem and forms an abstract *key* for it. The key is formed by replacing 1) all constants by 'constant', 2) all input variables by 'input-var', 3) all output variables by 'output-var', 4) each unary function *(F ?x)* by a binary function *(Attribute F ?x)*, and 5) all predicates (functions) by the abstract predicate (function) immediately above it in the abstraction hierarchy (fig. 1)

The reason for the fourth step is that we can view all unary functions as an abstract function, *attribute*, that takes two arguments - the name of the unary function and its parameter - and applies its first argument to the second. The fifth step abstracts predicates and functions by climbing one step up the abstraction hierarchy.

Using the above steps, the keys for the first constraint for problems 1 and 2 are (see fig. 1):

(contained input-var input-var), and *(subsumed input-var input-var)*

respectively, and the key for the second constraint for both the problems is:

(rel-operator (attribute constant input-var) constant)

Since the second key is identical for the two problems, the systematicity heuristic suggests that the two problems could be analogous.

Some care has to be taken when forming keys for predicates that are commutative or have a *commutative-dual*, defined as follows:

Definition: Let f and g be two binary functions. g is a commutative-dual of f if for all x and y, $f(x,y) = g(y,x)$.

Thus, the predicate $>$ is a commutative-dual of the predicate $\leq$, and *vice versa.* If all instances of an abstract predicate or function are commutative or have a commutative-dual, then the abstract operator is termed commutative. While forming keys, we need to ensure that for commutative operators, the key is not sensitive to the order of the arguments. For example, in problem P2 above, the second constraint could have been written as:

(> ?t (cpu-time ?p))

This does not change the essential nature of the problem and we want to treat both of them analogously. Therefore we define a *canonical form* to represent predicates, using the order of a predicate [3]. The canonical form is determined by permuting the arguments of all commutative predicates so that they appear in decreasing order (with variables preceding constants). Thus, the canonical form for the above predicate is:

(Rel-op (Attribute constant input-var) constant)

The system first converts all keys to a canonical form before using them for storing or retrieving problems.

3. Similar syntactic structure heuristic A third clue which can suggest that two problems might have analogous solution is the structural similarity of the problem definition. Thus, if both problems are defined recursively then their solutions might be similar. For example a program to find ancestors might consist of a shell file called 'ancestor'. When this file is called with an argument, the shell script checks whether the argument is '/' or not (end-of-recursion test); if it is '/', it does nothing (end-of-recursion-commands), otherwise it prints the parent of the argument and recurses by executing the same file again with the parent as the argument. Having solved this, it is easy to determine a program for descendants: The program would involve creating a similar shell file, that checks whether its input argument is a file or not; if it is a file, the program outputs the file; otherwise it recurses by executing the file again with each sub-directory under it as an argument.

Similarly, consider functions that are asynchronous. Their specifications are of the form: <command> :WHEN <condition>
where :WHEN is a keyword recognized by the system. Such programs involve waiting and checking periodically for the completion of some event.

To detect such analogies, the system analyzes the problem specification and identifies syntactic features like the presence of recursion, or certain keywords indicating the nature of the program - asynchronous, periodic, etc. These are then used to index and retrieve problems.

4. Argument abstraction heuristic. As in the case of functions, we can also make use of the abstraction hierarchy for objects which appear as arguments in the problem specification. In some cases if two problems have arguments that belong in the same abstraction hierarchy, the two problems may be analogous. For example, *count number of paragraphs, count number of lines* and *count number of words* are analogous. All involve finding a way of recognizing a text-object by finding its terminator (white space for a word, a newline

[3]The order is defined as follows: Constants and variables are order 0. The order of a predicate is 1 plus the maximum of the order of its arguments.

for a line, a blank line for a paragraph), mapping them to a countable object and counting that object. However, in other cases it may not work, e.g. a page is not recognized by a delimiter but in terms of the number of lines (typically 24).
Interaction of the heuristics: In general each of the above heuristic will suggest several, and possibly different, problems as a potential analogue of the target problem. Therefore the retriever needs to decide what importance should be attached to each suggested alternative, and in what order they should be tried.

The algorithm used by the analogue retriever works by retrieving all analogues using the systematicity heuristic and choosing the one that has the maximum number of indices pointing to it. If there is only one such analogue, it is returned as the best match. If no or more than one analogues are found, the functional, the syntactic and then the argument abstraction heuristics are used in a similar manner to break the conflict.

If there is no analogue retrieved by any of the three heuristics, the retrieval algorithm returns a failure. If there are still multiple analogues, one of them is returned arbitrarily.

4.2 ELABORATION: REPLAY OF PLANS

Once an appropriate source analogue has been determined for a target problem, the second stage of analogy begins, whereby the derivation of the source solution is used to derive an efficient solution for the target problem. We illustrate the derivational analogy method through an example. Suppose we want to count the number of sub-directories that are descendants of a given directory. The specification of this problem is similar to the example in section 3, except the post-condition, which is stated as follows:

```
(= ?n (card (SET (?sd :directory) :SUCH-THAT (descendant ?sd ?d))))
```

Let's assume that the heuristics defined earlier retrieves the count-files program as the closest analogue for this problem.

APU first checks whether there is a direct solution (i.e. a Unix command) for the problem. If it does not find a direct solution, it tries to apply the top-level rule in the source analog. There are three possibilities:

1) The rule is applicable with the same substitution of parameters, i.e. the corresponding variables in the target and source problems have the same argument type. Then, the two problems are identical and the entire sub-tree below the source analogue is copied (with the appropriate variable substitutions).

2) The rule is applicable with a different substitution of parameters. Then, the analogical reasoner applies the rule to the target problem. In general, this rule application would result in a decomposition of a problem into sub-problems $s_1, s_2, ..., s_m$ for the source and sub-problems $t_1, t_2, ..., t_n$ for the target. The algorithm attempts to solve sub-problems

$t_1 \ldots t_{min(m,n)}$ first, by analogy using sub-problems $s_1 \ldots s_{min(m,n)}$, and if any of them remains unsolved, by calling the planner. When $n > m$, the problems $t_{m+1} \ldots t_n$ are also attempted using the planner. If any of the sub-problems $t_1 \ldots t_n$ remains unsolved, the algorithm returns a FAILURE.

3) The rule is not applicable. APU then checks to see whether any of the other rules stored at the source node is applicable. If any of them are, then they are tried in turn till one of them returns a successful sub-plan for the problem. If none of the rules results in a complete solution, then the analogy algorithm calls the planner.

The application of the above algorithm results in the derivation shown in fig. 3. The analogy algorithm is able to successfully replay the steps of the original algorithm till it comes to the problem of listing descendants of a directory. At this point the analogy fails, and the planner is called. The planner tries to find another analog to solve the problem and may either retrieve a subroutine that lists all descendants of a directory or synthesize a program to list this using the rulebase. In the given example, we had already synthesized a program to list descendants of a directory and APU's retrieval heuristics succeed in retrieving that program for the current sub-problem. Thus that solution is used directly without having to resynthesize it. The other sub-problem for which the original command is not applicable is that of selecting directories from a list of directory-objects. APU searches for and finds a direct command (*grep "^d"*) to select directories. The resultant solution is:

```
< code to list descendants > | grep "^d" | wc -l
```

5 DISCUSSION

Currently, APU's rulebase consists of rules for a small subset of the Unix commands, dealing mostly with the manipulation of files, directories, text-objects, etc. using operations like counting, sorting, selecting, etc. Using these, APU has been used to successfully synthesize several (more than 40) programs. Most of these programs are analogous (to varying degrees) with one or more other programs and APU was able to synthesize them faster using replay. Some typical examples of programs generated using analogy include a program to find the most frequent filename in a system by analogy to a problem of finding the most common word in file, a program to kill all processes running for a specific period of time by analogy to a program for deleting all files with a specific size, and a program to list all ancestors of a directory by analogy to a program to list all descendants of a directory.

An important feature of APU is that the retrieval of source analogs for a problem is done automatically and the search for source analogs is done at various levels of a plan

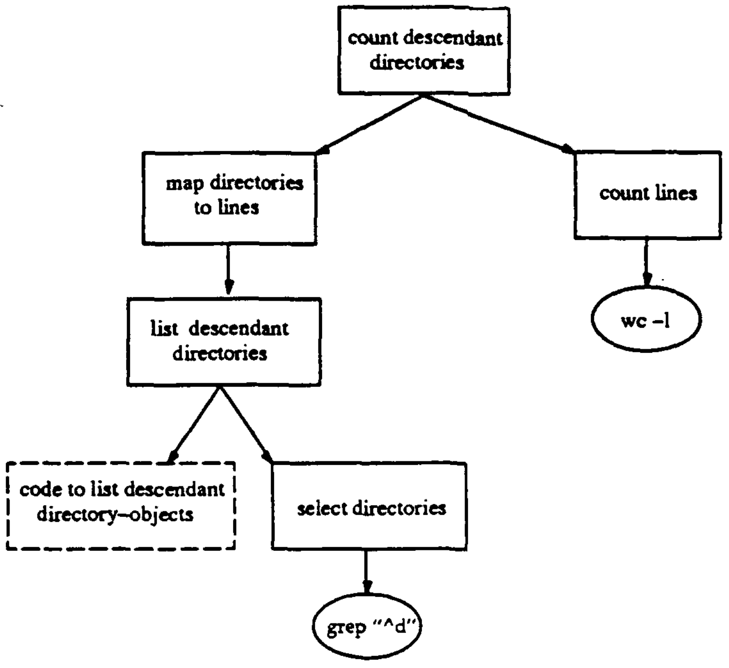

count descendant directories
map directories to lines
count lines
wc -l
list descendant directories
code to list descendant directory-objects
select directories
grep "^d"

Figure 3. Derivation of program to count sub-directories that are descendants of a directory

development. Thus APU is able to use analogy from several sources to synthesize a single program. Moreover, since it searches for direct commands to solve a sub-problem before applying analogy, it can sometimes improve upon a previous solution for an analogous problem. This partially alleviates a drawback in earlier solution transformation systems that depend solely on analogical transformations without taking into account alternative means for solving a problem.

6 RELATED WORK

The program development philosophy incorporated in APU is similar to that proposed in the KBSA paradigm [13], albeit on a much smaller scale and for a specialized domain. In particular, we adopt an evolutionary view of program synthesis, in which a program evolves from a specification by stepwise refinement, reusing previous derivation histories. In the KBSA paradigm, the emphasis is on design iteration: changes are made to a previously developed specification and replay is performed using the stored derivation of the older version of the same specification. APU extends this idea by attempting to use the derivation of other programs which are analogous to a current problem.

The use of analogy in program synthesis has been studied by several other researchers (e.g. [8,12,18,24,27]). The concept of derivational analogy was first formalized in [7]. A related approach called explanation-based case-based learning is recently receiving much attention [15].

The transformational program synthesis technique, on which APU is based, has a long history (e.g. [1,2,22]). Our work is also related to the extensive literature on software reuse that emphasize *generative systems*, advocating the reuse of design [5,6].

An excellent analysis of the issues involved in the reuse of design plans is provided in [16]. The systems described in that paper ([14,17,23,25]) and work being done at the Kestrel Institute ([12]) are perhaps the closest in spirit to our work. However, the emphasis in the above systems is more on design iteration and as such the problem of automatically detecting source analogues is finessed.

Finally, the idea of automating various aspects of Unix programming have been explored in [10,21,26].

7 CONCLUSION

We have described a system that records the derivation of programs and uses them to improve its performance on future analogous problems. The retrieval of the analogous problems is done automatically at all levels of program synthesis. The system has been tried on a small but diverse set of examples and we view the performance of the system, both in terms of retrieval and speed-up, as satisfactory. We plan to test our system in other domains besides Unix programming and evaluate its performance on more complex problems and a larger plan library.

References

[1] R. Balzer, N. M. Goldman, and D.S. Wile. On the transformational approach to programming. In *Second International Conference on software engineering*, pages 337–348, IEEE, 1976.

[2] David Barstow. An experiment in knowledge-based automatic programming. *Artificial Intelligence*, 12:73–119, 1979.

[3] Sanjay Bhansali and Mehdi T. Harandi. Apu: automating UNIX programming. In *Tools for Artificial Intelligence 90*, Washington, D.C., November 1990(to appear).

[4] Sanjay Bhansali and Mehdi T. Harandi. *Program Synthesis using Derivational Analogy.* Technical Report UIUCDCS-90-1591, University of Illinois at Urbana-Champaign, 1990.

[5] Ted J. Biggerstaff, editor. *Software Reusability, Vol. 1 : Concepts and Models*, ACM Press, New York, 1989.

[6] Ted J. Biggerstaff, editor. *Software Reusability, Vol. 2 : Applications and Experience*, ACM Press, New York, 1989.

[7] Jaime G. Carbonell. Derivational analogy and its role in problem solving. In AAAI, pages 64–69, 1983.

[8] N. Dershowitz. Programming by analogy. In Ryzsard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach, Volume II*, chapter 15, pages 395–424, Morgan Kaufmann Publishers, Inc., 95 First Street, Los Altos, CA 94022, 1986.

[9] Premkumar Devanbu, Ronald J. Brachman, Peter G. Selfridge, and Bruce W. Ballard. LaSSIE: a knowledge-based software information system. In *12th International Conference on Software Engineering*, Nice, France, March 1990.

[10] Stephanie M. Doane, Walter Kintsch, and Peter Polson. Action planning: producing UNIX commands. In *Proc. of the Eleventh Annual Conference of Cognitive Science Society*, pages 458–465, August 1989.

[11] D. Gentner. Structure-mapping: a theoretical framework for analogy. *Cognitive Science*, 7(2):155–170, 1983.

[12] Allen Goldberg. *Reusing software developments.* Technical Report, Kestrel Institute, Palo Alto, California, 1989. Draft.

[13] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich. *Report on a knowledge-based software assistant.* Technical Report RADC-TR-83-195, Rome Air Development Center, August 1983.

[14] M.N. Huhns and R.D. Acosta. *Argo: An analogical reasoning system for solving design problems.* Technical Report AI/CAD-092-87, MCC, Microelectronics and Computer TEchnology Corporation, Austin, TX, 1987.

[15] Janet L. Kolodner, editor. *Case-based reasoning workshop*, DARPA/ISTO, Morgan Kaufmann, Clearwater Beach, Florida, 1988.

[16] Jack Mostow. Design by derivational analogy: issues in the automated replay of design plans. *Artificial Intelligence*, 40:119–184, 1989.

[17] Jack Mostow, Michael Barley, and Timothy Weinrich. Automated reuse of design plans. *International Journal for Artificial Intelligence and Engineering*, 4(4), October 1989.

[18] Jack Mostow and Greg Fischer. Replaying transformationals of heuristic search algorithms in DIOGENES. In *Proc. of the* AAAI *Spring symposium on* AI *and software engineering*, Palo ALto, CA, March 1989.

[19] U. S. Reddy. Functional logic languages, Part I. In *Graph Reduction*, pages 401–425, Springer-Verlag, 1987. (Lecture Notes in Computer Science, Vol 279).

[20] Charles Rich and Richard C. Waters. *Readings in Artificial Intelligence and Software Engineering.* Morgan Kaufmann, Los Altos, California, 1986.

[21] Peter G. Selfridge. How to print a file: an expert system approach to software knowledge representation. In *AAAI-88*, pages 380–385, AAAI, St. Paul, Minnesota, August 1988.

[22] D. Smith, G. Kotik, and S. Westfold. Research on knowledge-based software engineering environments at kestrel institute. IEEE *Trans. on Software Engineering*, 11(11):1278–1295, Nov. 1985.

[23] L. I. Steinberg and T. M. Mitchell. The redesign system: a knowledge-based approach to VLSI CAD. IEEE *Design Test*, 2:45–54, 1985.

[24] J. W. Ulrich and R. Moll. Program synthesis by analogy. In *Proceedings ACM Symposium on Artificial Intelligence and Programming Languages*, pages 22–28, Rochester, NY, 1977.

[25] D. S. Wile. Program developments: formal explanations of implementations. *Communications of the* ACM, 26(11):902–911, 1983.

[26] R. Wilensky, Y. Arens, and D. Chin. Talking to UNIX in english: an overview of UC. *Comm. of the ACM*, 27(6):574–592, June 1984.

[27] Robert. S. Williams. Learning to program by examining and modifying cases. In *Proceedings Case-based reasoning workshop*, pages 463–474, Clearwater Beach, Florida, May 1988.

Formal Methods and Reusable Software

Shiu-Kai Chin
Department of Electrical and Computer Engineering & CASE Center
Syracuse University
Syracuse, New York 13244-4100
Telephone: (315)443-3776   email: chin@cat.syr.edu

Abstract – Software reuse is "the reapplication of a variety of kinds of knowledge about one system to another similar system in order to reduce the effort of development and maintenance of that other system", [2]. Software is essentially a "construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions", [3]. The challenge of building software is "the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation", [3]. Higher order logic and functional and logic programming languages, i.e. formal methods, provide a means for documenting design knowledge in an executable and formally verified manner which supports the reuse of design knowledge and the production of quality software. This paper describes some formal methods and tools which can be applied to the engineering of reusable software.

1 INTRODUCTION

Software reuse is one method of leveraging past work to produce new and larger systems. In its simplest and most narrow form, reusable software is a library of subroutines available to different programs. What is actually shared or reused between the various programs is the code and as such the reusable components are typically small. In a larger sense, software reuse means the reapplication of design knowledge, to create another system, [2]. In this case, what is shared is not code but concepts, e.g. parameterized programs, [5]. Various programs created using the same design concepts would expectedly have similar properties and hopefully be known to be correct according to some correctness criteria, [4].

Before delving into the details of formal methods, we will motivate their use by reviewing the essential qualities of software and the goals of reuse. After the review, the remaining sections will focus on several examples of increasing complexity which use formal development methods to create several verified and reusable pieces of software.

1.1 Essential Qualities of Software

Fred Brooks is known as the "father of the IBM System/360 computer family" and was the project manager for the Operating System/360 software. Based on his extensive software development experience, Brooks made the following observations on the essential nature of software and the intrinsic difficulties associated with its creation, [3].

> "The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions."
>
> "... the hard part of building software [is] the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation."

This work was sponsored by RADC Contract F30602-87-C-0215 and by the CASE Center

In other words, what is important about software is not the actual code, but the conceptual logic and solution structure the code represents.

1.2 Reuse Means More than Reusable Code

Given Brooks' view that concepts rather than lines of code are important, the notion of reusable software must include the notion of reusable concepts. What are reusable concepts? Essentially, reusable concepts are abstractions of objects. The abstractions hide the implementation details and focus on properties, e.g. behavioral descriptions and *relationships between inputs and outputs*.

This concept-based interpretation of reuse is not inconsistent with the more narrow view of software reuse as reusable code. Typically, when constructing a program which depends on using a matrix inversion routine for example, the programmer's interest in the inversion code is usually limited to conceptual issues such as correct functionality, accuracy, precision, and speed of execution rather than the precise details of data structures and control flow. So long as the actual and specified behavior is represented by the abstraction M^{-1}, the code is relatively unimportant.

1.3 We Are Not Alone

Software developers are not alone in their desire to reuse previous designs in newer ones. Hardware developers have coped with enormous increases in transistor integration by reusing designs and design knowledge. Current integrated circuits have close to one million transistors on a single chip. Ten years ago ten thousand was typical. The Intel 4004, 8008, 8080, 8086, 8088, 80186, 80286, 80386, and 80486 microprocessors are examples of design reuse in hardware development. Each new generation of Intel processors incorporates and extends the programmer's model and hardware design of the previous generation. The important point is each new generation is not built from scratch, but reuses previous work, albeit previous work with many modifications and extensions.

1.4 Keys for Design Reuse

The keys to design reuse are:

- Design specification.
- Design verification.
- The ability to compose designs like functions.
- Higher order designs.
- Computer-aided design tools.

What we will do in the remainder of this paper is illustrate the use of the above keys.

2 DESIGN SPECIFICATION AND VERIFICATION

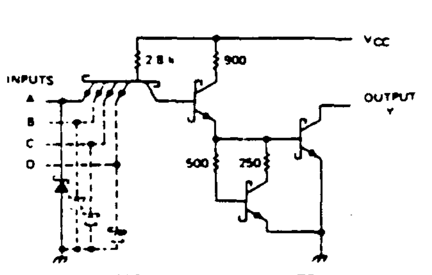
One reuse paradigm which has existed since the early 70's is TTL (transistor-transistor logic) design based upon the Texas Instruments 7400 MSI (medium scale integrated) circuit family. In fact, creating designs using the family is called "TTL-design" and is often used to characterize design styles which utilize prepackaged parts like standard macro-cells.

What makes TTL-design work is the TTL Data Book, [11]. More precisely, the physical book itself is not important, but the functional abstractions of the transistor circuits and the transistor-level implementations contained inside the book are.

For example, Figure 2.1 shows the transistor-level description of a 74S05 inverter. While containing only three transistors, one diode, and four resistors, it still has a fair amount of electrical information accompanying it, namely current levels and switching times as shown by the tables next to the circuit schematic.

TYPE	TEST CONDITIONS*	t_{PLH} (ns) Propagation delay time, low-to-high-level output			t_{PHL} (ns) Propagation delay time, high-to-low-level output		
		MIN	TYP	MAX	MIN	TYP	MAX
'01, '03	C_L = 15 pF, R_L = 4 kΩ for t_{PLH}, 400 Ω for t_{PHL}		35	45		8	15
'05			40	55		8	15
'12, '22			35	45		8	15
'H01, 'H05, 'H22	C_L = 25 pF, R_L = 280 Ω		10	15		7.5	12
'L01, 'L03	C_L = 50 pF, R_L = 4 kΩ		60	90		33	60
'LS01, 'LS03, 'LS05, 'LS12, 'LS22	C_L = 15 pF, R_L = 2 kΩ		17	32		15	28
'S03, 'S05, 'S22	C_L = 15 pF, R_L = 280 Ω	2	5	7.5	2	4.5	7
	C_L = 50 pF, R_L = 280 Ω		7.5			7	

*Load circuits and voltage waveforms are shown on pages 3-10 and 3-11.



'S03, 'S05, 'S22 CIRCUITS

Resistor values shown are nominal and in ohms.

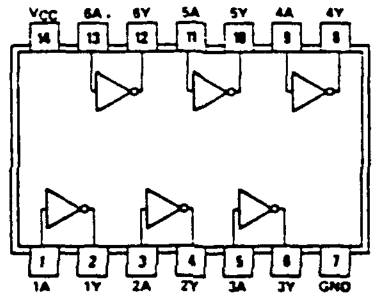
Figure 2.1 Inverter Implementation

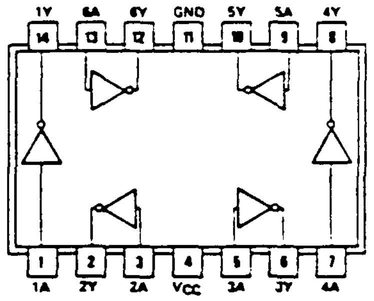
While the implementation description in Figure 2.1 is quite detailed, what is more often used by designers is the *specification description* of the 74S05 shown in Figure 2.2. Specifically, the relationship between output Y and input A as given by Y = ~A, and the precise identification of inputs and outputs.

HEX INVERTERS
WITH OPEN-COLLECTOR OUTPUTS

05

positive logic:
$Y = \overline{A}$





See page 6-4

SN5405 (J) SN7405 (J, N)
SN54H05 (J) SN74H05 (J, N)
SN54LS05 (J, W) SN74LS05 (J, N)
SN54S05 (J, W) SN74S05 (J, N)

SN5405 (W)
SN54H05 (W)

Figure 2.2 Inverter Specification

The underlying factor which made TTL-design so powerful was the implicit knowledge that for any object in the TTL Databook, that object's implementation and specification descriptions were the "same" or "equivalent". Designers could use the specified behavior in their designs and rarely, if ever, have to consider the physical transistor-level implementations of the TTL objects themselves.

The interchange of specifications and implementations is used constantly in VLSI and underpins the entire standard cell approach to design. It also is one of the underpinnings of hierarchical design.

The lesson for software design is similar: precise specifications of behavior are useful and necessary to manage complex designs successfully. Designers must have confidence that in some sense a reusable object's specification and implementation are the same.

3 FUNCTIONAL COMPOSITION AND HIGHER ORDER DESIGNS

If we continue to examine TTL-design as a successful instance of reusability, we find that another key to its success is functional composition and the implicit support for higher order inputs and outputs, i.e. sub-system components or modules can be created and debugged separately and then joined or "glued" together to form the whole system.

For example, if the inner product were to be computed, it would be reasonable to first compute all the product terms in one sub-system or module and sum them in another module.

Of course, modularity is not a new idea in software. Modularity is cited as one of the chief differences between structured and unstructured programs. Modularity brings productivity improvements as small modules are coded quicker than large programs, general purpose modules are reusable, and modules can be tested independently.

Nevertheless, underneath modularity is some notion of how components can be *combined* or *composed* to form new components and programs. As Hughes says in [7]:

> "When writing a modular program to solve a problem, one first divides the problem into sub-problems, then solves the sub-problems and combines the solutions. The ways in which one can divide up the original problem depend directly on the ways in which one can glue solutions together."
>
> "One can appreciate the importance of glue by an analogy with carpentry. A chair can be made quite easily by making the parts – seat, legs, back and so on – and sticking them together in the right way. But this depends on the ability to make joints and wood-glue. Lacking that ability, the only way to make a chair is to carve it in one piece out a solid block of wood, a much harder task."

Hughes goes on to say that functional languages have an advantage over conventional programming languages since functions are higher order, i.e. functions can accept other functions as input and can return functions as values, and functions or programs can be composed to form new functions or programs.

Hughes's view is that functional composition and higher-order functions are the glue – i.e. attributes which support functional abstraction and reusability. The utility of these attributes is exemplified by a program written in a higher order and parameterized style which sums lists of numbers. We note that this style is supported by the "D" or "declarative" programming language, [8, 9], developed for RADC/COES.

First, the notion of a list is defined by:

```
listof α ::= [] | α:(listof α)
```

which means that a list of αs is either empty, denoted by [], or is a non-empty list constructed from an α and another list of αs, where α names the set of objects from which the list elements are drawn, e.g. natural numbers, booleans, etc. The function which actually constructs a list given an element α and another list is denoted by ":" and is pronounced "cons". For convenience, bracket notation for finite lists is used where list elements are enclosed by "[" and "]" and separated by ",". Thus,

```
[1]       means 1:[]
[1,2,3]   means 1:(2:(3:[]))
```

The notion of summing a list of numbers is expressed recursively by:

```
sum [] = 0
sum (num:list) = num + (sum list)
```

That is, the sum of an empty list is zero, while the sum of a non-empty list of numbers is the number at the head of the list added to the sum of the remaining elements of the list.

While the above definition satisfies the problem at hand, it is an instance of a more general procedure which recursively applies a function to a list of elements. A more general and higher order "glue" function can be written called reduce which has two parameters, ⊕ – the binary operator applied to the list elements, and x – the value returned by reduce (⊕) x when applied to an empty list.

```
(reduce (⊕) x) [] = x
(reduce (⊕) x)(el:list) = el ⊕ ((reduce (⊕) x) list)
```

The sum function is implemented by:

```
sum = reduce (+) 0
```

While using reduce may seem like extra work, consider specifying the function product as the function which multiplies all its list elements together:

```
product [] = 1
product (num:list) = num * (product list)
```

We can reuse reduce to implement product by:

```
product = reduce (*) 1
```

Returning to our inner product example, if the inner product terms are represented by a list of pairs, e.g. [(x0,y0);(x1,y1); ... ;(xn,yn)] where "," denotes the infix pair operator and FST(x,y) = x and SND(x,y) = y, then the function product_pairs can be defined as:

```
product_pair [] = []
product_pair (pair:list) = ((FST pair)*(SND pair)):(product_pair list)
```

The inner product program inner_product is just the composition of product_pair and sum:

```
inner_product = sum o product_pair
```

where "o" denotes functional composition, i.e. (sum o product_pair) list = sum (product_pair list).

While the above examples are simple, hopefully they illustrate in spirit how higher order parameterized functions and functional composition support reuse by glueing together other functions.

Several functional languages exist including ML, Miranda, and Haskell, as well as our own declarative language "D" [8, 9] developed for RADC/COES. In particular, D not only supports Haskell functions, but supports Prolog Horn clauses as well.

4 VERIFICATION USING CAD TOOLS

To continue with Hughes's carpentry metaphor, once we have the parts and the glue, we have to put them together in the "right way". While documenting programs in the declarative style shown above is helpful, documenting the properties of programs is essential for reuse. After all, how can something be reused confidently if its function is unknown or unverified?

This is where formal descriptions of specifications and implementations and tools supporting formal reasoning like theorem-provers come in. They do the book keeping necessary to establish program properties and provide a cumulative knowledge-base on engineering design objects.

Machine-executable and machine-verified specifications and implementations are keys for managing design complexity through reuse. In [4], one of the major objectives of CAD for VLSI design is asserted to be:

"... [the] increase [in] the level of abstraction of machine-based design knowledge to keep pace with corresponding increases in circuit integration. A formally verified and machine interpretable knowledge base enables designers to operate at higher levels of abstraction with confidence."

To illustrate how verification is done, we offer two examples.

4.1 A First Order Example

Returning to our previous definition of sum, we may wish to formally prove that the *specification* of sum,

```
sum [] = 0
sum (num:list) = num + sum list
```

is in fact equivalent to its *implementation*, i.e.

```
sum = reduce (+) 0
```

Thus, we would want to prove the theorem *"for all lists, sum list is equal to reduce (+) 0 list"*, i.e.

|– ∀list. sum list = reduce \$+ 0 list

where "Γ |– t" means given the (possibly empty) list of logical terms Γ is true, then t is also true.

While the proof of this theorem is straightforward and is easily done by hand, to manage the complexity of design, the proof and the theorem must be *machine-executable and machine-verifiable* to make the definitions and properties of those definitions available and *reusable* by other designers and in later parts of the design.

As an example of how this is done, we show a sample automated proof using the Higher Order Logic (HOL) proof assistant created by Gordon, [6], which is one of the theorem provers used by the U.K. Ministry of Defense to verify *safety critical* designs. We note that HOL, as its name implies, is able to reason about higher order descriptions, e.g. descriptions where variables can be functions as well as primitive elements and functions can return functions as values. Thus, its reasoning power matches the descriptive power of the higher order languages in the previous section.

First, the definitions of sum and reduce are introduced as axiomatic definitions.

```
#let sum =
   new_recursive_definition false list_Axiom 'sum'
    "(sum [] = 0) /\ (sum (CONS (n:num) nlist ) =
      n + (sum nlist))";;

#let reduce =
   new_recursive_definition false list_Axiom 'reduce'
    "(reduce (f:num->num->num) (x:num) [] = x) /\
     (reduce (f) x (CONS el list) =
     (f el ((reduce (f) x) list)))";;
```

The definitions in higher order logic appear within the double quotes """. "CONS " denotes the list constructor ":" used in the previous section. "\$+" is the prefix version of +. "#" is the HOL system prompt. The theorem-prover function "`new_recursive_definition false list_Axiom 'reduce'`" means that the higher order logic expression which follows is 1) recursive, 2) is not an infix definition, 3) is based on the recursive structure of finite lists given by `list_Axiom`, and 4) should be assigned the name `reduce` in the database which makes up the definitions and theorems.

Next, we prove that the specification and implementation of sum are the same. First, the goal is specified.

```
#set_goal([],"∀list:(num)list.sum list = reduce ($+) 0 list");;
"∀list. sum list = reduce $+ 0 list"
```

The empty list in the set_goal line indicates that the theorem is to be proved with no other assumptions. To prove the goal for all finite lists, we use induction on lists to break the goal into its base case and inductive case.

```
#expand(LIST_INDUCT_TAC);;
OK..
2 subgoals
"∀h. sum(CONS h list) = reduce $+ 0(CONS h list)"
    [ "sum list = reduce $+ 0 list" ]

"sum[] = reduce $+ 0[]"
```

By rewriting the goal "sum[] = reduce $+ 0[]" using the definitions of sum and reduce, we can immediately prove the base case.

```
#expand(REWRITE_TAC [sum;reduce]);;
OK..
goal proved
|- sum[] = reduce $+ 0[]

Previous subproof:
"∀h. sum(CONS h list) = reduce $+ 0(CONS h list)"
    [ "sum list = reduce $+ 0 list" ]
```

We can also rewrite the inductive case using the definitions and then finish the proof by substituting the inductive hypothesis into the rewritten goal.

```
#expand(REWRITE_TAC [sum;reduce]);;
OK..
"∀h. h + (sum list) = h + (reduce $+ 0 list)"
    [ "sum list = reduce $+ 0 list" ]

#expandf(ASM_REWRITE_TAC []);;
OK..
goal proved
. |- ∀h. h + (sum list) = h + (reduce $+ 0 list)
. |- ∀h. sum(CONS h list) = reduce $+ 0(CONS h list)
|- ∀list. sum list = reduce $+ 0 list
```

Having proved that the specification and implementation of sum are the same, the definitions and the correctness theorem are available and, in a mechanical sense, "known" by the system and can be used and built upon by other designers or designs. In particular, for all inputs, the simpler specification can be substituted for the implementation.

We note here that virtually the same proof can be done showing that product is equivalent to its implementation reduce (*) 1. In fact, the same proof code can be reused with the appropriate substitutions: e.g. product for sum, etc.

4.2 A Higher Order Example

We mentioned previously the importance of functional composition and higher order functions and how they simplify the creation of larger programs. Functional composition and higher order functions also simplify the verification task. For example, consider the function APPEND which concatenates two lists together, e.g. APPEND [1,2,3] [4,5] = [1,2,3,4,5]:

APPEND [] list = list
APPEND (x:xs) list = x:(APPEND xs list)

We intuitively understand that with respect to sum and product the following statements are true:

∀list1 list2. sum(APPEND list1 list2) = (sum list1) + (sum list2)
∀list1 list2. product(APPEND list1 list2) = (product list1) * (product list2)

We could do two separate proofs as we did in section 4.1, but as both sum and product have the same underlying implementation in reduce, we instead prove a *higher order theorem* about reduce and specialize it to specific cases like sum and product. In effect, the verification of the above two properties is simplified by reusing the properties of reduce.

The property we prove about reduce is:

|- ∀f z l1 l2. (∀x. f z x = x) ∧ (∀a b c. f a(f b c) = f(f a b)c) ⊃
(reduce f z(APPEND l1 l2) = f(reduce f z l1)(reduce f z l2))

In other words, if z is the *identity element* for f – ∀x. f z x = x, and if f is *associative* – ∀a b c. f a(f b c) = f(f a b)c, then applying f to the reduction of the two lists is the same as reducing with f the two lists joined together – reduce f z(APPEND l1 l2) = f(reduce f z l1)(reduce f z l2).

We can prove the above using HOL system as follows. First, we strip-off the universally quantified variables f and z and use induction on l1.

```
#expand(STRIP_TAC THEN STRIP_TAC THEN INDUCT_THEN
       list_INDUCT ASSUME_TAC THEN REPEAT STRIP_TAC);;
#OK..
2 subgoals
"reduce f z(APPEND(CONS h l1)l2) =
 f(reduce f z(CONS h l1))(reduce f z l2)"
 [ "∀l2.(∀x. f z x = x) /\ (∀a b c. f a(f b c) = f(f a b)c) ⊃
    (reduce f z(APPEND l1 l2) = f(reduce f z l1)(reduce f z l2))" ]
 [ "∀x. f z x = x" ]
 [ "∀a b c. f a(f b c) = f(f a b)c" ]

"reduce f z(APPEND[]l2) = f(reduce f z[])(reduce f z l2)"
 [ "∀x. f z x = x" ]
 [ "∀a b c. f a(f b c) = f(f a b)c" ]
```

The base case, `reduce f z(APPEND[]l2)`, is proved by rewriting using the definitions of APPEND and reduce, and by rewriting using the assumption `["∀x. f z x = x"]`.

```
#expand(ASM_REWRITE_TAC [reduce;APPEND]);;
OK..
goal proved
. |- reduce f z(APPEND[]l2) = f(reduce f z[])(reduce f z l2)
```

The inductive case is proved by rewriting using the definitions of APPEND and reduce, and by rewriting using two terms in the assumption list we list here as A and B: A. `(reduce f z(APPEND l1 l2) = f(reduce f z l1)(reduce f z l2))` obtained by using the inference rule *Modus Ponens* on the conjunction of 1) ∀a b c. f a(f b c) = f(f a b)c and 2) ∀x. f z x = x with 3) ∀l2.(∀x. f z x = x) /\ (∀a b c. f a(f b c) = f(f a b)c) ⊃ (reduce f z(APPEND l1 l2) = f(reduce f z l1)(reduce f z l2)), and B. ∀a b c. f a(f b c) = f(f a b)c.

```
#expand(POP_ASSUM_LIST
 \thl.REWRITE_TAC[(MP(SPEC_ALL (el 3 thl))
                    (CONJ (el 2 thl)(el 1 thl)));
                  (el 1 thl)]);;
##OK..
goal proved
|- ∀f z l1 l2.
   (∀x. f z x = x) /\ (∀a b c. f a(f b c) = f(f a b)c) ⊃
   (reduce f z(APPEND l1 l2) = f(reduce f z l1)(reduce f z l2))
```

If we give the theorem the name reduce_APPEND, we can specialize it to the cases for sum and product as shown below. Notice that the preconditions now depend on showing that 0 and 1 are the identity elements for + and * and that + and * are associative.

```
#let th1 = SPECL ["$+";"0"] reduce_APPEND;;
th1 =
|- ∀l1 l2.
(∀x. 0 + x = x) /\ (∀a b c. a + (b + c) = (a + b) + c) ⊃
(reduce $+ 0(APPEND l1 l2) =
(reduce $+ 0 l1) + (reduce $+ 0 l2))

#let th2 = SPECL ["$*";"1"] reduce_APPEND;;
th2 =
|- ∀l1 l2.
(∀x. 1 * x = x) /\ (∀a b c. a * (b * c) = (a * b) * c) ⊃
(reduce $* 1(APPEND l1 l2) =
(reduce $* 1 l1) * (reduce $* 1 l2))
```

Fortunately, the HOL system already has the necessary theorems about + and *, so all that is necessary is to rewrite th1 and th2 with the theorems for + and *, and the previously proved theorems equating sum with reduce $+ 0 and product with reduce $* 1. The results are theorems th3 and th4.

```
#let th3 = REWRITE_RULE
          [ADD_CLAUSES;ADD_ASSOC;sum_theorem] th1;;
th3 =
|- ∀l1 l2. sum(APPEND l1 l2) = (sum l1) + (sum l2)

#let th4 = REWRITE_RULE
          [MULT_CLAUSES;MULT_ASSOC;product_theorem] th2;;
th4 =
|- ∀l1 l2. product(APPEND l1 l2) = (product l1) * (product l2)
```

Thus, we have proved the two desired properties by supplying the appropriate parameters to the higher order theorem relating reduce and APPEND.

5 BUILDING A FORMAL FRAMEWORK

In Section 4 we showed two simple verification examples. What we did was show the equivalence between different expressions where typically one expression could be thought of as a specification expression and the other as an implementation expression. As the examples were quite small, we were able to reason about them without using any additional structure.

We now consider a slightly larger example similar to [4] and [12] to illustrate the type of formal framework necessary for larger problems. The problem we consider is the creation of a computer-aided-design program which will correctly create an array of half adders to sum a column of bits. We would like to this program to 1) work for columns of arbitrary length, 2) be able to produce a wide variety of adder array structures, and 3) be "correct". What this example will show is the need to spend as much effort defining an appropriate framework to reason *about* the program as well as the program itself. Also, we will see the need to define precisely what "correct" means. This and other larger examples exist using the D language and the HOL system.

5.1 Basic Definitions

Figure 5.1 shows the basic half adder cell where a and b are the inputs for the half adder, and where So and Co are the sum and carry outputs.

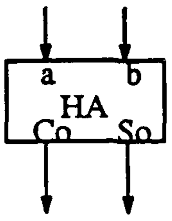


Figure 5.1: Half Adder Cell

Since we are using logic values to model numbers, we need to establish a mapping from the booleans to the natural numbers. We do this by defining a *value function* named BV which interprets the value of a bit.

$\vdash \forall bit.\ BV\ bit = (bit \rightarrow 1 \mid 0)$

The above notation means "for all bits, the expression BV bit can be replaced by bit → 1 | 0 and vice versa where "a → b | c " means "if a is true then b else c". Thus, BV maps T to 1 and F to 0 in the expected way.

Also, we need to explicitly state our interpretation of correct behavior. First, we define the outputs in terms of the inputs.

$\vdash \forall a\ b.\ HASUM(a,b) = {\sim}a \wedge b \vee a \wedge {\sim}b$

$\vdash \forall a\ b.\ HACARRY(a,b) = a \wedge b$

$\vdash \forall a\ b.\ HA(a,b) = HASUM(a,b),HACARRY(a,b)$

The above three formulas define the individual sum and carry functions and the entire half adder cell is viewed as a *pair*, where the first element of the pair is the sum output and the second element is the carry output. In fact, we can formally define the notion of what is meant by "the sum and carry outputs" of an adder by defining two accessor functions So and Co as follows:

$\vdash \forall x.\ So\ x = FST\ x$

$\vdash \forall x.\ Co\ x = SND\ x$

One way to view the half adder cell is as a *transformer*, i.e. a function which *rewrites* inputs in one form into outputs with a different but *equivalent* representation. This gives rise the the following theorem HACORRECT which can be proved by boolean case analysis on each input, a and b.

```
HACORRECT =
|- ∀a b. (BV a) + (BV b) = (BV(So(HA(a,b)))) + (2 * (BV(Co(HA(a,b)))))
```

What HACORRECT says is for all inputs a and b, the sum of their bit values equals the sum of the bit value of the sum output plus two times the bit value of the carry output. Notice that HACORRECT allows us to substitute the more complex expression (BV(So(HA(a,b)))) + (2 * (BV(Co(HA(a,b))))) involving the implementation function HA with a simpler expression of its behavior, (BV a) + (BV b), which does not contain the implementation function at all.

5.2 Intermediate Definitions

At this point, we have all of the *atomic* definitions and we can generalize them to larger operations and structures. First, we can define the value of a column of bits in terms of the individual bit values.

```
|- (COLVAL[] = 0) ∧ (∀h t. COLVAL(CONS h t) = (BV h) + (COLVAL t))
```

The value function COLVAL is primitively recursive. Its base case says the value of the empty column is 0 and its inductive case says that the value of a non-empty column is the bit value of the first element of the column, h, plus the column value of the remainder of the column, t. Notice that we could have defined COLVAL using the higher order function reduce and the MAP function we define next. Then we would have been able to deduce its properties whcn composed with other functions like APPEND, but for brevity we have used the simpler definition above.

The MAP function is a higher order function like reduce. It takes a function f and applies it to each element of a list.

```
|- (∀f. MAP f[] = []) ∧ (∀f h t. MAP f(CONS h t) = CONS(f h)(MAP f t))
```

From the definition above we see for example that MAP BV [T;T;F;T] = [1;1;0;1].

Using MAP we can generalize the application of HA onto lists of pairs.

```
|- ∀pairs.MAP_HA pairs =
   (let list = MAP HA pairs in
    let sumlist = MAP So list in
     let carrylist = MAP Co list in
   sumlist,carrylist)
```

Thus, the MAP_HA function takes a half adder and applies it to a list of input pairs and produces a pair of lists as output where the first list is the list of sum outputs and the second list is the list of carry outputs. For example, MAP_HA [(F,F);(F,T);(T,F);(T,T)] will produce internally: list = [(F,F);(T,F);(T,F);(F,T)]; sumlist = [F;T;T;F]; and carrylist = [F;F;F;T].

Since HA operates on pairs of input bits, for convenience we define a value function for pairs of bits.

```
|- ∀pair. PAIRBITVAL pair = (BV(FST pair)) + (BV(SND pair))
```

At this point we can prove the following theorem based on the definitions we already have.

```
MAP_HA_CORRECT =
|- ∀pairs.
  (COLVAL (FST(MAP_HA pairs))) + (2 * (COLVAL(SND((MAP_HA pairs))))) =
  (reduce $+ 0 (MAP PAIRBITVAL pairs))
```

Basically, the theorem says that the value of the sum column plus twice the value of the carry column equals the sum of all the values of the input pairs. It is proved by induction on the list pairs and by rewriting using the definitions.

5.3 Higher Order Design Functions

We now turn to generating interconnection structures for the array of half adders. We define a higher order function named HACOLTRANS which has as inputs: 1) split – a function which takes a column of bits and produces a pair of lists by splitting the column into two parts: a list of pairs produced from the first 2n elements of the column, and the remainder of the column, e.g. column [x0;x1;x2;x3;x4] could be split into ([(x0,x1);(x2,x3)],[x4]); 2) mergesum – a function which combines two sum columns; 3) mergecarry – a function which combines two carry columns; 4) sumcol – an input sum column; and 5) carrycol – an input carry column. What HACOLTRANS does is 1) split the input sum column using the splitting function and applies MAP_HA to the list of pairs; and 2) uses the sum and carry column combining functions to combine the remaining sum column with the sum column produced by MAP_HA, and the input carry column with the carry column produced by MAP_HA. The new sum and carry columns are returned as a pair.

```
|- ∀split mergesum mergecarry sumcol carrycol.
   HACOLTRANS split mergesum mergecarry sumcol carrycol =
   (let pairs = FST(split sumcol) in
    let restsum = SND(split sumcol) in
     let sumlist = FST(MAP_HA pairs) in
      let carrylist = SND(MAP_HA pairs) in
       let sumout = mergesum restsum sumlist in
        let carryout = mergecarry carrycol carrylist in
         sumout,carryout)
```

A correctness theorem for HACOLTRANS can be proved which says that so long as the functions split, mergesum, and mergecarry preserve the values of their inputs, then HACOLTRANS will produce a pair of columns equivalent in value to the input column pair.

```
HACOLTRANS_CORRECT =
|- (∀col.(reduce $+ 0 (MAP PAIRBITVAL(FST(split col)))) + (COLVAL(SND(split col))) =
        COLVAL col) ⊃
   (∀col1 col2.COLVAL(mergesum col1 col2) = (COLVAL col1) + (COLVAL col2)) ⊃
    (∀col1 col2.COLVAL(mergecarry col1 col2) = (COLVAL col1) + (COLVAL col2)) ⊃
  ((COLVAL (FST(HACOLTRANS split mergesum mergecarry sumcol carrycol))) +
   2*(COLVAL (SND(HACOLTRANS split mergesum mergecarry sumcol carrycol))) =
  (COLVAL sumcol) + 2*(COLVAL carrycol))
```

We can repeatedly apply HACOLTRANS n times as done by HACOLRED. If n = 0 then the pair (sumcol,carrycol) is returned. For n >0, HACOLTRANS is applied recursively to the input sum and carry columns.

```
|- (∀split mergesum mergecarry sumcol carrycol.
    HACOLRED 0 split mergesum mergecarry sumcol carrycol =
    sumcol,carrycol) ∧
   (∀n split mergesum mergecarry sumcol carrycol.
    HACOLRED(SUC n)split mergesum mergecarry sumcol carrycol =
    HACOLRED  n split mergesum  mergecarry
    (FST(HACOLTRANS split mergesum mergecarry sumcol carrycol))
    (SND(HACOLTRANS split mergesum mergecarry sumcol carrycol)))
```

A correctness theorem for HACOLRED can be proved saying that for all n, so long as the functions split, mergesum, and mergecarry preserve the values of their inputs, then HACOLRED will produce a pair of columns equivalent in value to the input column pair.

```
 HACOLRED_CORRECT =
 |- (∀col.(reduce $+ 0 (MAP PAIRBITVAL(FST(split col)))) + (COLVAL(SND(split col))) =
        COLVAL col) ⊃
   (∀col1 col2.COLVAL(mergesum col1 col2) = (COLVAL col1) + (COLVAL col2)) ⊃
    (∀col1 col2.COLVAL(mergecarry col1 col2) = (COLVAL col1) + (COLVAL col2)) ⊃
  (∀n.(COLVAL (FST(HACOLRED n split mergesum mergecarry sumcol carrycol))) +
    2*(COLVAL (SND(HACOLRED n split mergesum mergecarry sumcol carrycol))) =
   (COLVAL sumcol) + 2*(COLVAL carrycol))
```

5.4 Specific Designs as Instances of Specialized Higher Order Designs

All that remains now is to define some example splitting and merging functions and show the kinds of arrays they produce. We define two splitting functions. The first, HASPLITCOL1, splits a single pair off a column, the second, HASPLITCOL2, splits off as many pairs as is possible leaving at most one element in the remainder of the column. We use but don't define here the functions LENGTH, GETCOLPAIR, and STRIPCOL 2 to compute the length of a list, get the first two elements of a list, and strip off the first two elements of a list, respectively.

```
  |- ∀col.HASPLITCOL1 col =
     ((LENGTH col) < 2 → ([],col) |([GETCOLPAIR col],STRIPCOL 2 col))
```

Clearly, we can show by induction and rewriting that HASPLITCOL1 preserves the value of the input column as required by HACOLRED_CORRECT.

```
  HASPLITCOL1_CORRECT =
  |- ∀col.(reduce $+ 0 (MAP PAIRBITVAL(FST(HASPLITCOL1 col)))) +
     (COLVAL(SND(HASPLITCOL1 col))) = COLVAL col
```

The following function, RecHASplit pairs acc col, moves elements of a column col into an accumulator acc until two or more elements exist in acc. When acc has at least two elements, GETCOLPAIR is applied to acc to form a bit pair which is appended to pairs. The process is repeated recursively until no elements exist in the input column. Thus, RecHASplit [] [] [x0;x1;x2;x3;x4] = ([(x0,x1);(x2,x3)],[x4]).

```
  |- (∀pairs acc.RecHASplit pairs acc[] =
        ((LENGTH acc) < 2 →
         (pairs,acc) | (APPEND pairs[GETCOLPAIR acc],STRIPCOL 2 acc))) ∧
    (∀pairs acc bit bits.RecHASplit pairs acc(CONS bit bits) =
     ((LENGTH acc) < 1→
     RecHASplit pairs(APPEND acc[bit])bits |
    RecHASplit
     (APPEND pair [GETCOLAIR(APPEND acc[bit])])
     (STRIPCOL 2(APPEND acc[bit]))
      bits))
```

To simplify the application of RecHASplit, HASPLITCOL2 is defined which just initializes pairs and acc to empty lists.

```
  |- ∀col. HASPLITCOL2 col = RecHASplit[][]col
```

By induction and rewriting HASPLITCOL2 can also be shown to be correct.

HASPLITCOL2_CORRECT =
|- ∀col.(reduce $+ 0 (MAP PAIRBITVAL(FST(HASPLITCOL2 col)))) +
(COLVAL(SND(HASPLITCOL2 col))) = COLVAL col

All that remains to be done is to specify the functions used to merge the sum and carry columns. APPEND will be one function and CONCAT will be another where CONCAT uses the reverse order that APPEND does to join two lists.

|- ∀l1 l2. CONCAT l1 l2 = APPEND l2 l1

Clearly, both CONCAT and APPEND preserve the values of their input columns.

5.5 Design Examples

As examples, we now show two different arrays generated by using different splitting and merging functions. Both operate on the column [x0;x1;x2;x3;x4]. The first array shown in Figure 5.2a was created by the function HACOLRED 4 HASPLITCOL1 CONCAT APPEND [x0;x1;x2;x3;x4] []. This linear array corresponds to carry-save arrays often used because of its regular interconnection structure. The second tree array in Figure 5.2b was created by HACOLRED 3 HASPLITCOL2 APPEND APPEND [x0;x1;x2;x3;x4] []. As all the splitting, merging and reduction functions are correct, both arrays are also correct. Other array structures can be created using different splitting and merging functions.

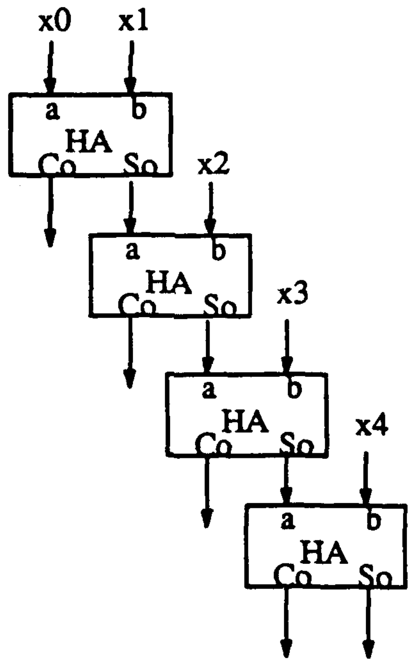


Figure 5.2a: Linear Array

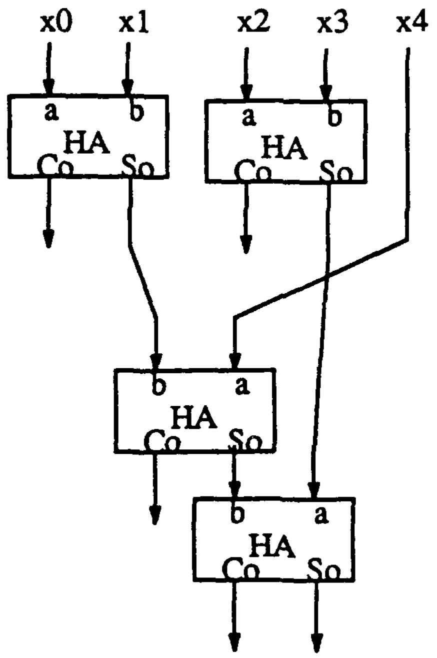


Figure 5.2b: Tree Array

As Figures 5.2a and 5.2b show, radically different but provably correct results can be obtained using the same underlying function, HACOLRED. This illustrates how design knowledge can be reapplied from one design to another or how general design knowledge can be specialized to specific instances. HACOLRED and its correctness theorem HACOLRED_CORRECT, by virtue of their parameterized nature, are extendible to other designs.

6 CONCLUSIONS

Design specification, verification, mathematical "glue" like functional composition and higher order parameterization, and CAD tools were cited as keys or enabling techniques for design reuse. Declarative languages like D developed for RADC, and theorem provers like HOL illustrated one set of realizations of these techniques. Declarative languages like D offer the functional "glue" and higher order parameterizaton missing from conventional languages and an execution semantics based on formal mathematics and logic. Theorem provers like HOL offer an associated means for recording and reasoning about software in a machine-executable fashion.

Additional work needs to be done to integrate the various tools and techniques which support specification, verification, and implementations. Specification languages based on formal logic, like Z, [10], are not executable and need to be "animated", i.e. made executable so as to observe the behavior of a specification. Translations from specification languages like Z to declarative and mathematical and logic-based languages like D offer one possible solution. Reasoning about specifications and implementations requires that the formal semantics of the specification and implementation languages be known and that their mechanizations also be correct. This has been partially done for D where the operational semantics of the abstract machine which mechanizes D have been formally verified with respect to the reduction rules of the language, [9].

Finally, some "meta-logical CAD tool glue" is also needed to integrate the CAD tools themselves. The MCC SpecTra environment [1] in which specification languages like Z, declarative languages like D, and theorem provers like HOL, are integrated in a hyper-object-based environment, may provide the necessary glue which enables users to move freely between specifications, animations, implementations, and proofs.

References

[1] J. Babcock, S. Gerhart, K.J. Greene, T. Ralston, SpecTra: A Formal Methods Environment, MCC Technical Report ACT-ILO-STP-324-89, August 1989.

[2] T.J. Biggerstaff, A.J. Perlis editors, *Software Reusability – Volume 1 Concepts and Models*, ACM Press, 1989, p. xv.

[3] F.P. Brooks, "No Silver Bullet - Essence and Accidents of Software Engineering", *IEEE-Computer*, April 1987, pp. 10-19.

[4] S-K Chin, "Verified Synthesis Functions for Negabinary Arithmetic Hardware", *Applied Formal Methods for Correct VLSI Design*, Luc Claesen editor, Elsevier, 1989.

[5] J.A. Goguen, "Principles of Parameterized Programming", *Software Reusability – Volume 1 Concepts and Models*, T.J. Biggerstaff, A.J. Perlis editors, ACM Press, 1989.

[6] M.J.C. Gordon, "HOL: A Proof Generating System for Higher-Order Logic", *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P.A. Subrahmanyam editors, Kluwer, 1988.

[7] J. Hughes, "Why Functional Programming Matters", *The Computer Journal*, vol. 32, no. 2, 1989, pp. 98–107.

[8] D.A. Jamsek, K.J. Greene, S-K Chin, P.R. Humenn,"WINTER: Wams IN Tim Expression Reduction", *Proceedings of the North American Logic Programming Conference*, Cleveland, Ohio, October 16-19, 1989.

[9] D.A. Jamsek "The WINTER Architecture: Support for a Purely Declarative Language", CASE Center Technical Report, No. 9007, Syracuse University, May 1990.

[10] J.M. Spivey, *The Z Notation – A Reference Manual*, Prentice Hall International, 1987.

[11] *The TTL Data Book – 2nd Edition*, Texas Instruments Inc., 1976.

[12] S-K Chin, "Synthesis of Arithmetic Hardware Using Hardware Metafunctions", *IEEE Trans. CAD*, Vol. 9, No. 8, pp. 793-803, Aug. 1990.

Persistence with Integrity and Efficiency*

Deborah A. Baker, David A. Fisher and Frank P. Tadman

Incremental Systems Corporation
319 South Craig Street
Pittsburgh, Pennsylvania 15213 U.S.A.
(412) 621-8888

Abstract

Managing persistent data efficiently and conveniently is a difficult task. Persistent object sizes vary from a single bit to many megabytes. The placement of objects in both main memory and secondary storage must be controllable by tools. The integrity of the objects must be maintained while allowing concurrent access. Operating system file systems, databases and software development systems provide partial solutions to the safe, efficient management of persistent data. An Iris-based information management system provides solutions to key persistent data problems heretofore solved only partially, unsatisfactorily, or inefficiently including type integrity for application-defined types, and safe data identification mechanisms.

1 Introduction

There have been many advances in research and development of technology, tools, and environments for design, implementation, and maintenance of large scale software applications during the past 20 years. Even though many of these component technologies are demonstrably effective for some limited aspect of the software process, there has not been any practical way for them to work cooperatively.

*This work was supported in part by the Rome Air Development Center under contract F30602-88-C-0115, the Defense Advanced Research Projects Agency (Arpa Order 5057) monitored by the Department of the Navy, Space and Naval Warfare Systems Command under contract N00039-85-C-0126 and by the Defense Advanced Research Projects Agency (Arpa Order 6487-1) under contract MDA972-88-C-0076.

In this paper we present a conceptual design and an instantiation of a suite of mechanisms that enable sharing and communication of information among the tools and tool components populating a (possibly distributed) software environment. The mechanisms ensure type and object integrity of all persistent information without advance knowledge of their types. They provide the primitive mechanisms required for the higher level imposition of user-defined policies such as those for version control, configuration control, release control, and access control.

In Section 2, the problem is described in more detail, along with the requirements placed on solutions. Section 3 contains a discussion of our model for providing persistence. In Section 4, an Iris instantiation of our model is discussed. Finally, in Section 5, our findings are reviewed and some future work is outlined.

2 Nature of The Problem

Some data may outlive the invocation of the tool or program which created them, in which case they are said to be *persistent*. Persistent data contain information which is important to an application taken as a whole, and which may be needed by several components (or invocations of components) of the application. There are several important requirements for persistent data in distributed software development, *maintenance* and operational environments that have not been addressed by databases or by file and operating systems. These requirements are concerned primarily with the integrity and efficiency of storage and retrieval of information within such environments.

In software environments, the persistent data obviously includes the requirements, designs, specifications, implementations and execution results of the programs being developed. It also includes representations of those programs in the form of source text, internal representations, unlinked object code, and executable target code. The persistent data also includes artifacts from, and inputs to, analysis, documentation, testing, project management, and maintenance processes such as constraints, rules, histories, and decision trees. All of these data items may exist simultaneously in a variety of versions and configurations. It is also crucial in software environments that the various tools and tool generators of the environment themselves be data objects of the environment.

Integrity for the data in software environments requires that all data be strongly typed with the type protection enforced throughout the persistent object base, not only for a few built-in types, but for those defined later by application developers and by tool builders as well. As in any distributed system, distributed software development, maintenance and operational environments must provide safe mechanisms to either maintain consistency among multiple copies of data objects that are replicated throughout the distributed system or to detect inconsistency..

Efficiency is critical to the feasibility of any system including distributed software environments. Efficiency issues arise primarily in three areas: delay in accessing data, efficiency in handling inter-object references, and the cost of maintaining the type, version, and configuration integrity of the persistent data. For instance, for our compilation systems, the semantic analyzer must process 3,000 lines per minute if the compiler is to process 1,000 lines per minute. The semantic analyzer for the Ada programming language will make approximately 25,000 access of small grain objects while processing 3,000 lines.

Traditional solutions, whether drawn from operating systems, databases, or software development environments, have typically been inefficient and seldom safe. They assume that the primary location of data does not change, that cross-reference among objects is infrequent or is the responsibility of the user, that type integrity is the responsibility of the user, or that inconsistency can be tolerated when the number of resulting erroneous computations is statistically low.

The Knowledge Based Software Assistant Program is exploring the use of a formally based paradigm, which involves mediation from the software assistant, in the full range of activities associated with software development and maintenance. The functionality associated with each activity is captured in a *facet*, or sub-assistant.

The *information* that is involved in the KBSA paradigm to date (by virtue of the facets under development) includes such things as requirements, specifications, design history, assumptions, reasons, and rationales[Ele89]. Another vital aspect of the KBSA paradigm is the reuse of this information [Gol89]. For instance, the requirements and specifications developed under their respective facets should be reused by the program development facet to assure that they are satisfied and by the project management facet to assist in scheduling and cost estimation. The most effective way to facilitate this cooperation among the KBSA facets via sharing and reuse of information is to provide efficient mechanisms whereby the information can be identified, stored, accessed, maintained, shared and reused in a distributed environment. It is exactly a substrate of this nature that is the subject of this paper.

Extant Partial Solutions Traditional programming languages, operating systems and databases have each addressed some aspects of persistent information management, but each has shortcomings with respect to our requirements.

Operating systems address problems of resource management and security, providing mechanisms and policies for allocating and sharing basic computing resources. Of concern here are the storage systems provided by operating systems (i.e., file systems); they do not typically ensure type integrity. This is true for both abstract and representation types and for invocations of tools as well as manipulations by human users. Furthermore, operating systems rely on user-defined names, which compromise identity integrity.

Databases capture some knowledge of the characteristics of the information they manage, and exploit that knowledge to make better use of resources. This knowledge is represented in a variety of ways (e.g. schemata and dependencies) and is used to optimize representation and access to information, to improve the convenience, reliability, and efficiency of maintaining important relationships between items of information, and to drastically reduce the time required to design and implement applications. The success of databases depends on certain assumptions such as there are relatively few schemata and relatively many items per schemata, the schemata are fixed, or change only infrequently, the types of information are closely circumscribed (for instance, relation is not a type in a relational database and therefore a relation can not itself participate in a relation) and they are not dynamic (i.e., types cannot be added arbitrarily), and the granularity of the information is known and fairly uniform. Information in traditional databases is most often distinguished on the basis of some key; such value-oriented names compromise identity integrity.

Unfortunately, the underlying assumptions of operating systems and databases are not valid for the information in a software environment. Software development environments are characterized by a wide variety of objects, with dynamically varying types and relationships. Correct and effective management of these objects requires intimate knowledge of the policies and relationships which are specified (implicitly or explicitly) in the objects themselves. The types of information in the system at any time are specified by that information, and the number of items of information of any given type may range from one to millions. Moreover, some objects are virtual, and are only instantiated dynamically, by applying one body of information to another.

Related research includes databases [Ber87], persistence [AB87, BB87, Coo87], software development environments [SDE88, CAI88, TBC+88], and object orientation [KC86, Mey89]. Much of this work is relevant in many ways. However, we have not entirely accepted the requirements, implicit or explicit, of these other projects, and there are, consequently, significant differences between most of the other projects and our own.

The models in object oriented systems are quite specific (e.g., a regime in which the data in an object include methods for responding to messages); furthermore, these systems are typically single user and/or single machine and the efforts to allow sharing among users and distributed machines are not altogether satisfactory. The use of a persistent programming language or a database programming language may be fine, but is not a useful approach for organizations that have mandates to use specific languages, or for organizations that have pre-existing software that they wish to use with as little modification as possible. Unlike many of the research projects on persistence, our system has requirements for strong typing and against restrictions on the types of persistent data. Also, research projects in these areas have, of necessity, concentrated on attainment of functional requirements, often at the expense of performance or reliability. Successful commercial systems have typically achieved their performance goals by focusing on important but limited application domains.

3 A Model for Providing Persistence

Ensuring integrity and enabling efficiency are design features of our substrate for the management and sharing of information. We present a three layer conceptual model. The first layer provides general mechanisms for large- and small-grain object management. The next layer is implemented in terms of the first and provides attributed information structures. The third layer is Iris: a special kind of attributed information structure that instantiates the design.

Integrity Error-proneness of traditional operating and file systems and databases arises because data can be referenced only by symbolic name, directory structure location, or value. In particular, it is impossible to guarantee *integrity of reference* as the physical location of data changes and to retain integrity of reference to data located on removable media. Our solution provides location-independent internal names that uniquely identify each data object.

The mechanisms developed for this project provide and maintain an *identity* for each type and each data object. The requirements for these identities are dictated by the nature of persistent data. The identity of an object must be *unique* to avoid confusing it with other objects. The identity of an object must be *universal* (i.e., must never change) to avoid invalidating the knowledge of an object's identity in one part of the system when a change is made elsewhere. The identity of an object must be *location-independent* because the location of an object may change in ne course of its lifetime.

Integrity is a pervasive goal of an integrated software environment and *type integrity* is central to persistent object management. It matters little how good other aspects of a system are (i.e., how fast it runs, or how much it encompasses), if it produces results that are incorrect or unreliable. Because types are used to express the formal properties of data, object management must include enforcement of the typing mechanisms to ensure integrity. Software applications use such a wide variety of data that it is impossible or impractical to build the complete spectrum into their persistent data system; they are forced to map their types onto the few supported by a database, with loss of integrity and increased error-proneness as the result.

The mechanisms developed for this project support an open-ended type system in which types can be added at any time and in which individual type properties need not be known to the persistent data system. However, the only way to guarantee type integrity for a piece of information is to have absolute control over all manipulations of the information. This includes determining exactly what operations can be applied to the piece of information. Partial type integrity can be provided when data is being manipulated by the object management mechanisms and by requiring that users of a piece of data have knowledge of its type.

Efficiency Obviously, many characteristics vary with an object's granularity. Examples include expected frequency of access, the complexity and nature of interrelationships with other objects, lifetime, flexibility of the access function set, and the performance requirements on the access functions. The most successful current object managers utilize granularity-based knowledge to tune the overall system performance. As an example, consider the problem of providing complete control over small-grained objects. Allowing them to be independently placeable, independently identifiable, and controlling access to them on an individual basis would be prohibitively expensive and unnecessary for most applications. Different mechanisms, then, are appropriate for different levels of granularity. Careful selection of appropriate granularity for the persistent data of an application is essential in achieving performance.

Current state-of-the-art approaches to object management all distinguish between large and small-grained objects. The distinction is not simply size. Large-grained objects are independent entities, whereas related small-grained objects are grouped into collections which are often represented as a single large-grained object. A file system may be viewed as a structure of large-grained objects (files). Each file is composed of small-grained objects (records or characters), in a certain organization scheme. Object management systems attempt to support these types of relationships in a more general manner.

Conceptual Model The remainder of this section will outline the conceptual model upon which the Iris mechanisms for providing persistence efficiently and with integrity are based.

Figure 1 shows object management from an Iris perspective. The vertical dotted line shows the division between large-grained and small-grained object management components. The two horizontal dotted lines separate general purpose object management (the lowest level), attributed information structures (in the middle), and Iris based persistence (the highest level).

At the lowest level, an *object manager* (left) provides a general set of operations needed to manage large-grained objects and each *item manager* (right) implements a particular

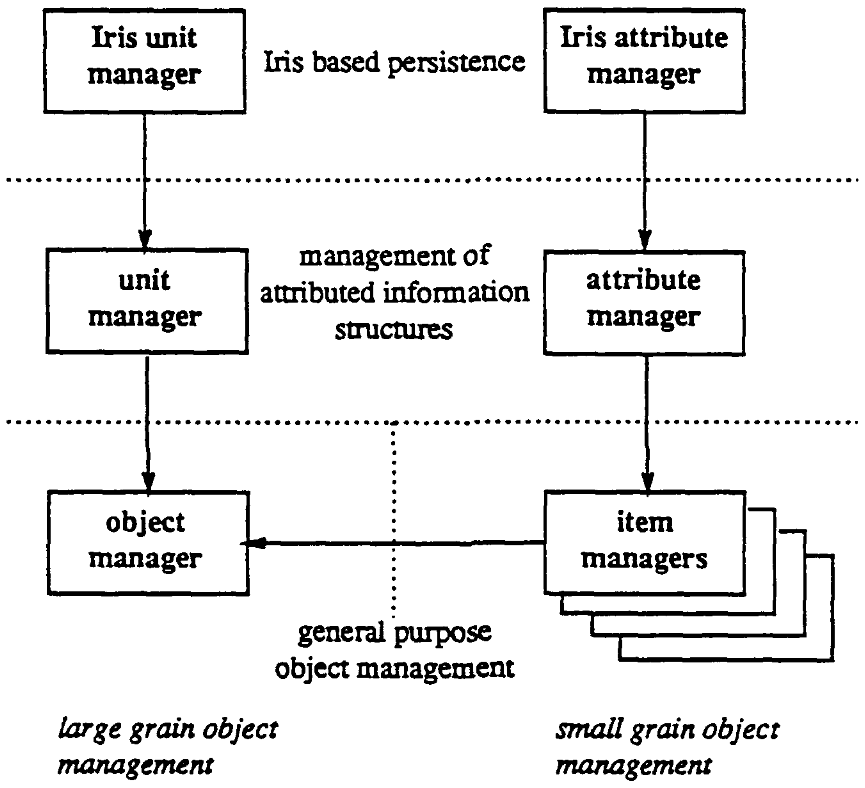

Iris unit manager
Iris based persistence
Iris attribute manager
unit manager
management of attributed information structures
attribute manager
object manager
item managers
general purpose object management
large grain object management
small grain object management

Figure 1: Model of Iris Based Persistence

form of small-grained object management. The item manager exports a data abstraction, the *segment*, which serves as a container for an indexed collection of small-grained objects called *items*. A segment is itself a large-grained object. Operations on (large grain) objects include translations between peripheral storage and main memory. The operations on items include fetch, store and space allocation, all within a segment. Objects (i.e., segments) are independently placeable and identifiable. Items are uniquely identifiable within a segment.

Items can be organized into collections (i.e., segments) in a variety of ways, depending on the properties of the attributed information structure they represent. The designer of a tool must be able to choose a representation for the attributed information structure that exhibits storage utilization and item access times that are appropriate for the application. Item managers vary in the organization of items in a segment because attributed information structures vary in characteristics such as attribute density, attribute size, and uniformity of attribute size.

An *attributed information structure* is a collection of *entities* and information, called *attributes*, about them. The middle level in Figure 1 corresponds to the management of attributed information structures. An entity is a carrier of information. Each entity has an identity and a set of attributes. Each attribute of an entity is a piece of information relevant to the entity; taken as a whole, the attributes of an entity contain all the information concerning it. A *unit* is a collection of entities. The concepts at this second level are built upon those at the lower level.

Every entity is a member of an *entity type*. An entity type includes a list of *attribute definitions*. Each attribute definition specifies the name and value type of an attribute of members of that entity type. If the type of a particular entity has an attribute definition, the entity may or may not actually have that attribute. If it does not, the attribute is said to be *missing*. When a new attribute definition is added to an entity type, the attribute defined is missing for all existing entities of that type, but can be added to some or all members of that type by appropriate attribute manager operations.

Tools do not need to know the entire set of definitions of an entity type, but only about those that are relevant to the function of the tool. A tool's *view of an entity type* is therefore a subset of attribute definitions contained in that entity type. The attribute manager should support views in such a way that changes to an entity type should affect only those tools which have a view in which that change is visible; tools which have a view in which the change is not visible should not require modification or even recompilation.

An entity collection is an indexed collection of entities of the same type. Entity collections are independently placeable and identifiable. Notice that both objects and entities have unique, universal, location independent identity. Objects are an implementation mechanism; their identities serve to distinguish various chunks of physical storage. Entities are an abstract mechanism; there is no single chunk of storage that corresponds to an entity. Their identities also serve to distinguish them.

The highest level provides Iris-based persistence, discussed in Section 1. Iris unit and Iris attribute management are instantiations of the more general unit and attribute management at the attributed information structure level.

4 Iris Based Persistence

The data objects of a software environment include both composite and atomic objects. They can therefore be thought of as utterances in various formal languages. The languages represented include implementation, specification, design, requirements, prototyping, process programming, and constraint languages. Iris[1] provides solutions to the information managements problems of distributed software environments. It is a semantically based system for representing and managing pieces of information that can be viewed as utterances in some formal language(s). An Iris system includes a common information structure as well as both small- and large-grain management. Iris based persistence has been used in an Ada-to-Iris tool, where Iris unit management is equivalent to Ada program library management. An analysis of this design combined with measurement data on earlier prototypes indicates that performance in excess of 50,000 item accesses per second on a Sun 3/60 is achievable. This is well within the performance goals set out in Section 2.

The Iris Information Structure. The Iris information structure is a language independent form for representing the sentences of any formal language. It serves as a medium of information exchange and sharing among the tools of a software environment. It is an extensible and open-ended system with respect to the information it can capture and represent.

At an abstract level, the Iris information structure is a tree. Each Iris tree represents a composition or expression consisting of *references* and *applications*. Corresponding to this, an Iris tree is composed of two kinds of nodes: *reference nodes* and *application nodes*. For example, the expression $f(x, g(y, z))$ consists of references to entities named f, x, g, y, and z and applications of f and g. Reference nodes are interpreted as references to declarations that appear elsewhere in the Iris structure. The first child of an application node is its *operator*. The operator identifies an *operation* which is applied to the remaining children, which are called *arguments*. Frequently, the operator is a reference to the declaration of a named operation, but it can be any operation-valued expression represented as an Iris tree.

[1]Iris was the Greek goddess of the rainbow and messenger of the gods.

If the reference nodes of Iris are viewed as leaves (terminals) then the Iris representation can also be viewed as an abstract syntax tree with the application nodes acting as nonterminals. Each reference node does contain, however, a reference to a declaration which is itself an application node appearing earlier in (a preorder walk of) the Iris structure.

Iris is unique in that all operators are described within its own structure. It has no primitives. This means that individual tools need recognize and provide special case processing for only those operations that related directly to the functionality of the tool. For example, a semantic analyzer need recognize only operations that are declaration, scope, or type valued but does not have to distinguish between control structures and arithmetic operations. This contributes to the simplicity and small size of Iris based tools.

Iris is also a higher order system in that it provides full support for computed operations. A computed operation may appear either in place at the point of its application (i.e., as another application node which is the operator of the application) or as the value of a declaration which is referenced at the point of call (i.e., as a reference node which is the operator of the application). The combination of internal and higher order specification means Iris can be used to represent any formal language and that Iris based tools can be reconfigured for multiple and evolving languages with little or no change to their components.

To specify the representation of any language L, two things are needed: a *grammar* and a set of *L-standard declarations*. The grammar describes the correspondence between the concrete syntax of the language and its abstract syntax represented as Iris expressions. The L-standard declarations specify the built-in operations of the language, i.e., those operations which are available within the language but are not declared within programs of the language (e.g., control structures in implementation languages or invariants in specification languages).

Iris Attributes: Small Grain Object Management Small grain object management in Iris includes item and attribute management. Iris trees are implemented as collections of attributes. Each node in an Iris tree, along with its attributes, is an entity. Each attribute value of each entity is an item.

Units: Iris Large Grain Object Management Figure 2 depicts an entity collection. Each row is an entity and each column is an attribute of the type of the entities in the collection (i.e., an attribute collection). Each square is an attribute of a particular entity in the entity collection, and is represented as an item.

There are two ways to group Iris attributes for storage, as shown in Figure 2. The first is to group all the attributes of a single entity into a unit. This is called *horizontal*

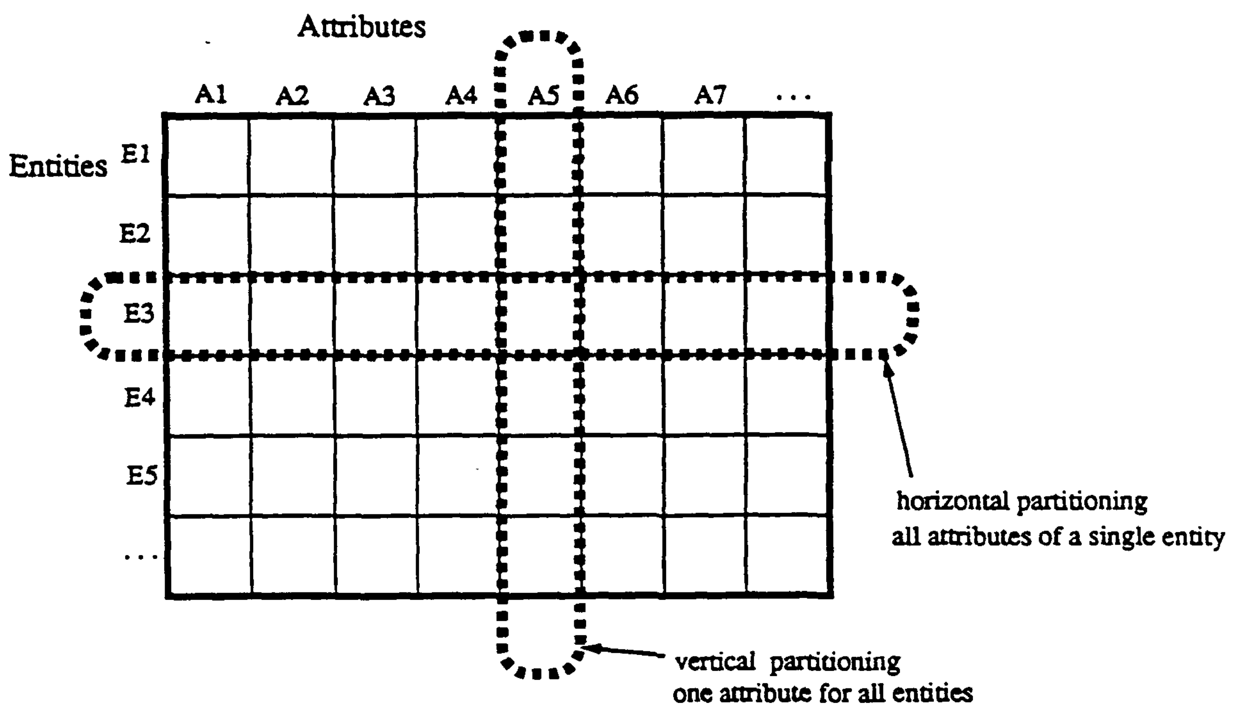


Figure 2: Organization of Iris Attributes for Storage

partitioning. The second is to group a single attribute for a collection of related entities into a unit. This is called *vertical* partitioning. While either partitioning is adequate, Iris uses vertical partitioning. The advantages of vertical partitioning over horizontal are twofold: new attributes can be added without impacting existing attributes or tools and attributes not needed by a tool need not be loaded into memory. The disadvantage of vertical partitioning over horizontal is that accessing attributes is more complex.

5 Summary and Future Work

Integrity and Inconsistency Information in a software environment is generated from many sources (some human interactions, some computations). It is frequently updated, and is subject to change from multiple, independent sources. Consistency of the information is difficult to maintain in such volatile situations. Furthermore, in distributed systems and in the presence of removable media, communication delays necessitate replication of frequently accessed data and preclude complete consistency among all copies.

There are several commonly used approaches that attempt to eliminate inconsistency by periodic, controlled update of changed data. The first, which might be called "lock the world", is simple in concept and is formally sound, namely: when it is necessary to update a data structure (or set of related data structures) place a lock on the entire database that will prevent both access and reference from any source other than the updating process until the update (of all copies) is complete. Such an approach, of course, can have extremely high cost in performance.

There are several variations that can significantly improve performance. The most obvious is to lock only those portions of the world that are directly affected by the update, thus allowing independent activity to continue in parallel. Another is to select a very small granularity of data for locking in order to maximize the number of independent parallel activities that can be accommodated simultaneously with an update. Such methods can work quite well in small, highly localized databases, but are prohibitively expensive in distributed systems because of the inherent communications delays.

An alternative which overcomes some of the communications delay is to partition the database itself into disjoint partitions (typically the nodes of the distributed network) and then to prohibit interpartition accesses. Locking and update can then be accomplished one partition at a time and propagated throughout the network. This approach reduces the delay as seen by any one node by accomplishing the communication outside the lock. This solution tends to increase the amount of mutable data that must be replicated and imposes a strict requirement for independently managed partitions. The former restriction complicates sharing of data stored at a central location; the latter precludes the use of removable media as a means of sharing and moving mutable data. In any case practical systems based on this approach have generally been those in which update propagation times of the order of one day are acceptable. It then is possible to do local updating within each partition during the day and propagate all the changes at night when there is little or no use of the systems.

The cost that cannot be tolerated in most systems is not the communications delays, but rather the delays imposed by locking. Solutions must either tolerate delays in general system response time caused by the locking, accept the one day update delays that are a consequence of updating only at night, or find a way to update without locking.

Locking, then, is prohibitively expensive and techniques to overcome these expenses are not uniformly applicable. While careful design reduces the adverse consequences of the remaining inconsistency, there are no guarantees against inconsistency nor even that adverse consequences are detectable.

The traditional method of updating without locking is simply to remove the lock for purposes of access or for both access and update. Other means are used to minimize the probability that erroneous or inconsistent data will be accessed, or that the adverse effects of such accesses will not be catastrophic. One of the better known examples of this approach are airline reservation systems in which there is a single shared central copy of the most current data. All updates must occur directly to this shared copy and are performed there under lock. Additional local copies are used for access purposes and can be arbitrarily far out of date. Similarly, update requests can be delayed (actually queued) arbitrarily long without delaying the updating process. The effect is that sometimes an access will indicate available seats but the update will fail because none are available, or a reservation will be accepted (i.e., queued for update on the assumption that it will succeed) and then later fail due to the effects of other queued requests. The latter results in overbooking.

We find all such approaches unsatisfying. Locking is, in general, prohibitively expensive in distributed systems and the techniques to overcome those expenses are not applicable in many situations. The traditional nonlocking approaches do not guarantee consistency or even detect it, but instead attempt to minimize adverse consequences. Our approach rejects as infeasible avoiding inconsistency. Instead, our approach creates a situation in which inconsistency is safely and efficiently detected and managed. The key to detection of inconsistent data is twofold. First, the various (updated) versions of an object must be distinguishable (universal identities are adequate for this purpose, see Section 3). Secondly, objects or application that are used in combination must maintain records of the identities of the versions or types of the objects with which they must interact. These techniques have been used successfully in our Ada to Iris tool in order to enforce order of compilation.

Summary The solutions outlined here can significantly improve the efficiency, reliability and robustness of applications that use persistent data. Key among these are distributed software development environments, such as that in the KBSA paradigm. The solution consists of efficient mechanisms that form a substrate to facilitate cooperation among KBSA facets via sharing and reuse of information. The substrate permits identifying, storing, accessing, maintaining, sharing and reusing information in a distributed environment.

Some of the current and planned scientific and engineering features that contribute to the efficiency and safety of our mechanisms for the management of persistent objects are:

- Object identity that is unique, location independent and universal.
- Integrity for all types, even those that are user defined, while the data is under the purview of the persistent mechanisms.
- Safe sharing via object identity rather than via user given names or data values.
- Recognition and management of inconsistency in the volatility of software development environments where data is frequently updated from multiple, independent sources.

- Vertical partitioning of attributes.
- Operations optimized for the different demands of small- and large-grain objects.
- Multiple item managers to exploit the various properties of attributed informations structures.

Acknowledgements

We thank our Incremental Systems colleagues David Mundie and Jonathan Shultis, both of whom were instrumental in the development of Iris. We acknowledge the Arcadia Consortium whose members have encouraged this work with their reviews, comments and suggestions.

References

[AB87] M. P. Atkinson and O. P. Buneman. Types and Persistent Database Programming Languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[BB87] P. Buneman and F. Bancilhon, editors. *Workshop on Database Programming Languages*, Roscoff, France, September 1987.

[Ber87] P. A. Bernstein. Database System Support for Software Engineering. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 166–178, Monterey, CA, March 1987. IEEE Computer Society.

[BFS87a] D. A. Baker, D. A. Fisher, and J. C. Shultis. A Practical Language to Provide Persistence and a Rich Typing System. In *Workshop on Database Programming Languages*, Roscoff, France, September 1987.

[BFS87b] D. A. Baker, D. A. Fisher, and J. C. Shultis. Persistence and Type Integrity in a Software Development Environment. In *Workshop on Persistent Object Systems: their Design, Implementation and Use*, Appin, Scotland, August 1987.

[BSF88] D. A. Baker, J. C. Shultis, and D. A. Fisher. Mechanisms for Providing Persistence in a Distributed Software Development Environment. In *Proceedings of the 3rd Annual Knowledge Based Software Assistant Conference*. Rome Air Development Center, August 1988.

[CAI88] U.S. Department of Defense, Proposed Military Standard DOD-STD-1838A (Revision A). *Common Ada Programming Support Environment (APSE) Interface Set (CAIS)*, 1988.

[Coo87] R. Cooper, editor. *Workshop on Persistent Object Systems: their Design, Implementation and Use*, Appin, Scotland, August 1987. Universities of Glasgow and St. Andrews.

[Ele89] D. M. Elefante. An Overview of RADC's Knowledge Based Software Assistant Program. In *Proceedings of the 4th Annual Knowledge Based Software Assistant Conference*. Rome Air Development Center, September 1989.

[Gol89] A. Goldberg. Reusing Software Developments. In *Proceedings of the 4th Annual Knowledge Based Software Assistant Conference*. Rome Air Development Center, September 1989.

[KC86] S. N. Khoshafian and G. P. Copeland. Object Identity. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA '86)*, pages 406–416. *SIGPLAN Notices*, 21(11), 1986.

[Mey89] N. Meyrowitz, editor. *Proceedings of the Object-Oriented Programming Systems, Languages, and Applications Conference (OOPSLA '89)*, New Orleans, Louisiana, October 1989. ACM SIGPLAN, *SIGPLAN Notices*, 24(10).

[SDE88] ACM SIGSOFT. *Third Symposium on Software Development Environments*, Boston, Massachusetts, November 1988. Appeared as *Sigplan Notices 24*(2) and *Software Engineering Notes 13*(5).

[TBC+88] R. N. Taylor, F. C. Belz, L. A. Clarke, L. Osterweil, R. W. Selby, J. C. Wileden, A. L Wolf, and M. Young. Foundations for the Arcadia Environment Architecture. In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, pages 1–13, Boston, MA, November 1988. ACM SIGSOFT/SIGPLAN. Appeared as *Sigplan Notices 24*(2) and *Software Engineering Notes 13*(5).

GRAPH-BASED LANGUAGE SPECIFICATION, ANALYSIS AND MAPPING WITH APPLICATION TO THE DEVELOPMENT OF PARALLEL SOFTWARE

Paul D. Bailor, Gary B. Lamont, and Thomas C. Hartrum

Department of Electrical and Computer Engineering
School of Engineering
Air Force Institute of Technology
Wright-Patterson Air Force Base, Ohio 45433
(513) 255-3708 or AV: 785-3708
e-mail: pbailor@galaxy.afit.af.mil

ABSTRACT — This paper presents a generalized formal language theory model used for the specification, analysis, and mapping of graphs and graph-based languages. The developed model is defined as a graph generative system, and a summary of the analysis results from a set theoretic, formal language, algebraic, and abstract automata perspective is presented. Additionally, the graph generative system model serves as the basis for applying graph-based languages to the specification and design of software. The specific application area emphasized is the use of graph-based languages as user-friendly interfaces for wide-spectrum languages that include structures for representing parallelism. The goal of this approach is to provide an effective, efficient, and formal method for the specification, design, and rapid prototyping of parallel software. To demonstrate the u·ility of the theory and the feasibility of the application, two models of parallel computation are used. A graph-based Petri net syntax is formally mapped into the corresponding linear syntax of a Communicating Sequential Processes (CSP) model of parallel computation where CSP is used as the formalism for extended wide-spectrum languages. Additionally, the Petri net to CSP mapping is analyzed from a behavioral perspective to demonstrate that the CSP specification behaves in a manner equivalent to the Petri net model.

1 INTRODUCTION

The basic premise of this investigation is that graph-based languages are a valid specification and design formalism for developing parallel software. Graphs are a fundamental mathematical concept, and their use pervades many aspects of the software development process and theoretical computer science. A few of the many areas in which graphs and graph theory have been successfully applied and which directly support the basic premise of this investigation are: software specification, software analysis, formal models of parallel computation, visualization of parallel processes, algorithm-to-architecture mapping techniques, process scheduling, load balancing, parallel programming languages, and compiler theory. Since graphs are a valuable visualization tool for representing relationships between entities and it is possible to formalize their representation in terms of formal language theory [1], it seems reasonable to hypothesize that formalized graphs and languages of graphs can be used as an important part of the process used to specify and design parallel software. Additionally, if the graphs can be made general enough to allow for the use of icons and complex labeling structures, the amount of syntactical and "semantically suggestive" information they represent can be significantly increased.

One of the most prevalent problems in the specification and design of parallel software is that the proposed methodologies have not kept pace with the development of parallel programming languages [2]. Several sequential programming languages have been extended to include parallel constructs and many new parallel programming languages have been developed. Unfortunately, much research still needs to be performed on the methodologies for developing parallel algorithms to better utilize the power of the explicitly parallel programming language constructs. Currently, the methodologies lack a good conceptual model of the process of parallel software design that is integrated with some formal model of parallel computation [3]. Some of the more well-known formal models of parallel computation are: finite state machines, computation graphs, Petri Nets, program schemata, algebraic models of communicating processes (e.g., Milner's Calculus of Communicating Systems and Hoare's Communicating Sequential Processes), concurrent objects, and UNITY. Additionally, the methodologies lack an associated *formal* description of the syntactic operations required to transform the statement of a computing problem into an effective and efficient parallel software system coded in an appropriate parallel programming language [4]. Therefore, the development of parallel software is currently a complex task because it is being driven more by the constructs contained in the target parallel programming language and/or the features of the target architecture than by a solid specification and design approach. Hence, what is desired is the ability to develop parallel software on a basis relatively independent of the architecture concentrating on the architecturally independent issues first and gradually introducing architecturally dependent issues as the level of detail increases.

One possible solution is the use of program transformation systems. In this approach, a wide-spectrum specification language is used to either explicitly or implicitly specify a parallel processing approach to the software system, and the parallel software is derived by successively applying transformation rules to obtain the software design and subsequent programming language code. Some work has been accomplished in the academic environment to develop a set of transformations for parallel systems. Manna and Wolper suggest the use of temporal logic to synthesize the derivation of concurrent processes [5]. Ladkin takes an algebraic approach called interval calculus to specify the behavior of concurrent processes [6]. The CIP-L wide-spectrum language incorporated a limited set of explicit constructs for parallel composition [7]. Alternately, the researchers studying Ada-based, wide-spectrum languages developed two separate languages [8]. While program transformation systems have shown promise, research in this area is only beginning. Additional research needs to be accomplished on developing the language constructs used by wide-spectrum languages to specify the potential for parallel behavior and on developing the transformations applied to th

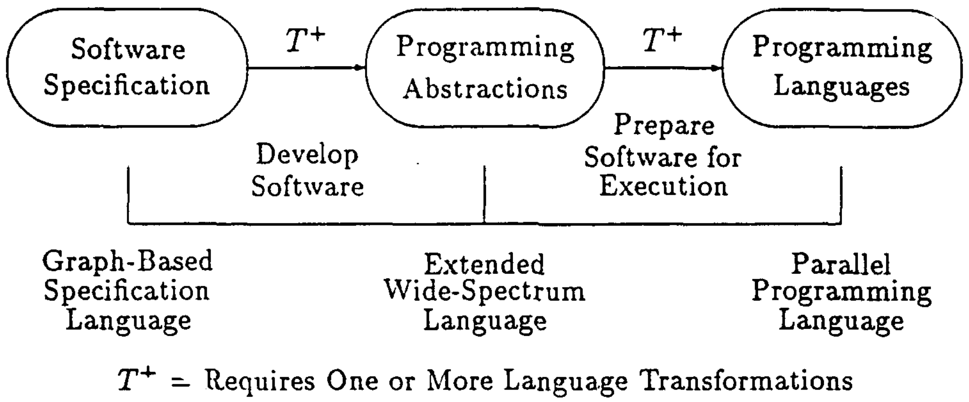

Software Specification
T^+
Programming Abstractions
T^+
Programming Languages
Develop Software
Prepare Software for Execution
Graph-Based Specification Language
Extended Wide-Spectrum Language
Parallel Programming Language
T^+ = Requires One or More Language Transformations

Figure 1: Graphical Language Interface Into a Wide-Spectrum Language

language constructs.

A potentially valuable approach for the specification, design, and rapid prototyping of parallel software is to use a formalized graph-based language as an interface into an executable wide-spectrum language extended with parallel processing constructs. Essentially, this means a set of formalized syntactical transformations can be developed which transform syntactical structures in the graph-based language into syntactical structures in the extended wide-spectrum language. The syntactical structures of the wide-spectrum language are then transformed into the syntactical structures of an appropriate parallel programming language. Figure 1 illustrates this approach. Based on this framework, the process of developing parallel software involves the use of a formalized, graph-based software specification language to initially specify the behavior of the parallel software system (Note that this implies a graph-based model of parallel computation is central to the approach). Next, the syntactical structures of the graph-based language are transformed into the syntactical structures of an extended wide-spectrum language. Thus, the specification can now be executed to verify its behavior. Last, the syntactical structures of the wide-spectrum language are gradually transformed into a formal design for the parallel software and finally into some target parallel programming language.

This paper does not address the transformations needed to go from the parallel programming constructs contained in a wide-spectrum language to the constructs contained in parallel programming languages. Additionally, this effort does not attempt to define new wide-spectrum languages. It is assumed that current generation, wide-spectrum languages can be extended to include parallel programming abstractions and constructs without a major redesign of the basic language structure.

In the remaining sections of this paper, a formal model of graphs and graph-based languages is defined and analyzed from several perspectives. Also, using the framework outlined in Figure 1, the model is applied to the task of specifying and designing parallel software. The graph-based specification language to be used is Petri nets and the explicit parallel processing constructs of CSP are used to extend wide-spectrum languages to the case of parallel software.

2 FORMAL MODEL OF GRAPHS AND GRAPH-BASED LANGUAGES

A formal language theory model of graphs and graph-based languages is defined in this section. The resulting model is called a graph generative system and implementing structures such as graph grammars are also defined. An extensive requirements analysis was conducted first to obtain a set of effective requirements for graph generative systems and graph-based, software specification languages. The requirements were determined based on the goals of this investigation as well as a review of the literature in the areas of graphical languages, software specification languages, and the intended application area of parallel software. The requirements as stated below are a synthesis over a wide range of numerous books and papers; therefore, no specific citations are given for any one of the requirements.

1. Requirements for Graph Generative Systems.

   (a) Needs to generate a large class of graphs and graph-based languages. For example, generate graphs of an arbitrary size, order, degree, connectivity, etc..

   (b) Needs to generate "complex" labels for the nodes and edges of the graphs as well as "complex" diagramming symbols. For example, it may be necessary to generate a set of attribute/value pairs for the nodes and edges of a graph. Such a set is more complex than the simple regular expressions needed to generate an identifying label of a node or edge. Similarly, it may be necessary to generate diagramming symbols (icons) more complex than the simple geometric shapes of circles, ovals, arcs, etc..

   (c) Needs to support more than one representation. That is, in certain cases the set theory representation of the graph may be desired over the multidimensional representation of a graph and vice versa.

   (d) Needs to be easily and efficiently applied by humans. That is, the steps used to produce/generate a graph are short and do not involve complex notation. Ideally, the generating systems should mimic the way humans draw graphs.

2. Requirements for Graph-Based Specification Languages.

   (a) Needs to show relationships important to parallel software, such as: data precedence, communication paths, temporal, hierarchical, and spatial.

   (b) Needs to be recognized, parsed, and mapped in an efficient amount of time and space.

   (c) Needs to be extensible in the sense that experienced systems analysts and programmers tend to perturb specification and design languages to meet their own desires.

   (d) Needs to include or be mappable into formal structures for software design and implementation.

Based on these requirements, existing models of graph generating systems were evaluated for their suitability as a generic, formalized model of graph generating systems. None of the existing models satisfied all of the desired requirements; therefore, a new model was constructed that builds on the previously developed theory. The definitions are based on a set theory approach to graphs; however, it is important to note that the definitions are not limited to describing only the set theoretic representation of graphs. In fact, the definitions provide only a mathematical description of the generating functions. Implementations for the generating functions are not explicitly described within the definition. Therefore, this model can be applied to multiple representations and implementation methods for graphs.

The starting point for defining a generalized model of graph generative systems is an examination of the alphabets associated with it. Two alphabets are used by the generalized model.

DEFINITION 2.1 *Let Δ be a finite, nonempty label alphabet used to construct the labels of the nodes and edges in the graph. The set Δ^* denotes the set of all possible labels constructable from Δ including the empty (or null) label Λ. Note that for any given graph, the set of labels contained within the graph will be a subset of Δ^*.*

DEFINITION 2.2 *Let Σ be a finite, nonempty diagramming symbol alphabet used to construct the diagramming symbols (icons) contained in the graph. The set Σ^* denotes the set of all possible diagramming symbols constructable from Σ including the empty (or null) symbol Γ. Note that for any given graph, the set of diagramming symbols contained within the graph will be a subset of Σ^*.*

The label and diagramming symbol alphabets are now used as the basis for defining a generalized model of a graph generative system.

DEFINITION 2.3 *A graph generative system G over the alphabets Δ and Σ is a 6-tuple $G \triangleq (\Phi, N, \Psi, E, \mathcal{R}, \mathcal{X})$ where*

(1) $\Phi : \Delta^* \times \Sigma^* \longrightarrow N$ is the node generation function for the node set N. Each node $n \in N$ has the form $n = (\delta, \sigma)$ where $\delta \in \Delta^*$ and $\sigma \in \Sigma^*$.

(2) $\Psi : N \times \Delta^* \times N \times \Sigma^* \longrightarrow E$ is the edge generation function for the edge set E. Each edge $e \in E$ has the form $e = (n_1, \delta, n_2, \sigma)$ where $n_1, n_2 \in N$, $\delta \in \Delta^*$, and $\sigma \in \Sigma^*$.

(3) $\mathcal{R}$ is a possibly empty set of relation generation functions where each $R \in \mathcal{R}$ is constructed using the cartesian product operation over a finite combination of subsets of N, E, Δ^*, and Σ^*. Formally, this is denoted $R \subseteq \prod_{j=1}^{n}(A_j)$ where $\forall j$, $A_j \subseteq N$, or $A_j \subseteq E$, or $A_j \subseteq \Delta^*$, or $A_j \subseteq \Sigma^*$.

(4) $\mathcal{X}$ denotes the set of sets generated by the application of the relation generation functions contained in $\mathcal{R}$.

(5) Remarks:

(a) Both the node generation function Φ and the edge generation function Ψ are relation generation functions meeting the criteria defined in item 3. However, both of these functions are basic to the mathematical definition of a graph and are not included in $\mathcal{R}$ for this reason.

(b) The functions Φ and Ψ are defined as independent functions. However, this does not imply that Φ and Ψ cannot be composed into a single function.

Based on Definition 2.3, two different perspectives can be taken. From one perspective, a graph generative system G generates a single graph with node set N, edge set E, and set of sets $\mathcal{X}$. The generated graph is denoted $G(N, E, \mathcal{X})$. The second perspective is that a graph generative system G generates many graphs where each graph g has a node set $N_g \subseteq N$, an edge set $E_g \subseteq E$, and a set of sets $\mathcal{X}_g$ generated by the functions in $\mathcal{R}$. In this case, g is denoted $g(N_g, E_g, \mathcal{X}_g)$. Thus, in the second perspective, N is the set of all possible nodes generated by G, E is the set of all possible edges generated by G, and $\mathcal{X}$ is the set of all possible sets generated by $\mathcal{R}$. Since the second perspective properly includes the first perspective, the second perspective leads to more generalized results. Therefore, this investigation uses the second perspective that a graph generative system actually generates a set of graphs as opposed to a single graph.

2.1 GENERALIZED DEFINITION OF A GRAPH GRAMMAR

Definition 2.3 provides no clue as to how the generating functions Φ and Ψ can be realized. This separation is intentional as it provides a great deal of flexibility; however, a mechanism for implementing the generating functions does need to be developed. One such mechanism is a graph grammar. Rather than provide specific examples of past research such as [9, 10, 11, 12, 13], a generalized definition of a graph grammar based on previous research is provided. In the simplest case, a graph grammar can be viewed as a generalization of the Chomsky grammars. A Chomsky grammar is typically defined as a four tuple $G = (V_N, V_T, P, S)$ where V_N is a finite set of variables, V_T is a finite alphabet of symbols, P is a finite set of productions, and $S \in V_N$ is a special variable called the start symbol. Graph grammars are different from a Chomsky grammar in the following ways. First, the sets V_N and V_T are augmented to allow for graph-based variables and symbols in addition to the normal string symbols. Second, the form of the production rules is modified to include the concept of an embedding transformation. In the case of graphs, the production rules become more complicated because they must specify how to embed the subgraph to be rewritten into the remainder of the graph. Therefore, different from string grammars, the production rules are a triple $P = (g_l, g_r, E_R)$ where g_l is the subgraph to be replaced (the left hand side of the production), g_r is the graph being inserted for it (the right hand side of the production), and E_R is the embedding transformation [1]. The embedding transformation is essentially a generalization of string concatenation to multidimensional structures, as in graphs. Additionally, the type of the embedding transformation is the main criterion for distinguishing various approaches to graph grammars, and research on embedding transformations has been one of the main research areas in the study of graph grammars [1].

A generalized definition of a graph grammar is presented next.

DEFINITION 2.4 *A graph grammar is defined as a five-tuple $G_G \triangleq (V_N, V_{T_L}, V_{T_D}, P, S)$ where*

(1) V_N is a finite, nonempty set of distinct symbols called variables, containing one distinguished symbol S, called the start symbol. Note that V_N can contain different types of

variables. For example, one type of variable for graph structures and one type for label structures.

(2) V_{T_L} *is a finite, nonempty set of terminal label symbols used to generate node and edge labels. Note that* V_{T_L} *is equivalent to the label alphabet* Δ *defined by Definition 2.1.*

(3) V_{T_D} *is a finite, nonempty set of terminal diagramming symbols used to generate node and edge diagramming symbols. Note that* V_{T_D} *is equivalent to the diagramming symbol alphabet* Σ *defined by Definition 2.2.*

(4) $V_N \cap (V_{T_L} \cup V_{T_D}) = \emptyset$.

(5) P *is a finite set of productions in which each production is of the form*

$$(g_l, g_r, E_R) \textit{ such that } g_l \rightarrow g_r \textit{ subject to } E_R$$
$$\textit{where } g_l \in (V_N \cup V_{T_L} \cup V_{T_D})^+ \textit{ and } g_r \in (V_N \cup V_{T_L} \cup V_{T_D})^*$$

A graph g is said to be generated or directly derived by the grammar G_G if there exists a sequence of productions over $(V_N \cup V_{T_L} \cup V_{T_D})^*$

$$g_0 \Longrightarrow g_1 \Longrightarrow g_2 \Longrightarrow \cdots \Longrightarrow g_n \textit{ where } n \geq 1$$

such that: 1.) g_0 is the start symbol S, 2.) $g_n = g$, and 3.) g_{i+1} is obtained from g_i, $0 \leq i \leq n$, by replacing (rewriting) some occurrence of a subgraph g_l (which is the left-hand side of some production in P) in g_i by the corresponding g_r (which is the right-hand side of that production) subject to the rules of the embedding transformation E_R for that production. The production rules in P are applied non-deterministically until the graph being generated, g, contains only terminal symbols in V_{T_L} and V_{T_D}. The derivation process for g is denoted by $S \overset{*}{\Longrightarrow} g$. The set of graphs generated by a graph grammar G_G is called a graph-based language and is denoted $L(G_G)$. Formally, $L(G_G)$ is defined as:

$$L(G_G) = \{g \mid S \overset{*}{\Longrightarrow} g \textit{ and } \forall A \in V_N, A \notin g\} \tag{1}$$

Similar to the Chomsky grammars, size constraints can be applied to the production rules of Definition 2.4 to arrive at type-0 (unrestricted), type-1 (context-sensitive), type-2 (context-free), and type-3 (regular) graph grammars and graph languages.

Definition 2.4 is now extended to accommodate the notion of a graph generative system as provided by Definition 2.3.

DEFINITION 2.5 *Let* $G_G = (V_N, \Delta, \Sigma, P, S)$ *be a graph grammar over the alphabet* Δ *and* Σ *with* P *containing production rules of the form* (g_l, g_r, E_R). *Extend* G_G *by partitioning the set of variables* V_N *and the set of productions* P *as follows*

1. Let $V_N = V_{GS} \cup V_{LS} \cup V_{DS}$ *such that* $V_{GS} \cap V_{LS} \cap V_{DS} = \emptyset$ *where*

(a) V_{GS} *is the set of variables used to generate the structure of the graph (i.e., the interconnection pattern of the nodes and edges),*

(b) V_{LS} is the set of variables used to generate the labels of the nodes and edges, and

(c) V_{DS} is the set of variables used to generate the diagramming symbols of the nodes and edges.

2. Let $P = P_{GS} \cup P_{LS} \cup P_{DS}$ such that $P_{GS} \cap P_{LS} \cap P_{DS} = \emptyset$ where

(a) P_{GS} is the set of production rules used to generate the structure of the graph (i.e., the interconnection pattern of the nodes and edges) where each production rule has the form (g_l, g_r, E_R). Additionally, the production rules can contain variables from any of the three subsets of V_N; however, the production rules cannot contain terminal symbols from the alphabets Δ and Σ.

(b) P_{LS} is the set of production rules used to generate the labels of the nodes and edges where each production rule has the form $\alpha \rightarrow \beta$. Each production rule can contain variables only from the subset V_{LS}. Additionally, each production rule can contain terminal symbols only from the alphabet Δ.

(c) P_{DS} is the set of production rules used to generate the diagramming symbols of the nodes and edges where each production rule has the form (d_l, d_r, E_R). Each production rule can contain variables only from the subset V_{DS}, and they can contain terminal symbols only from the alphabet Σ.

Thus, Definition 2.5 allows for the generation of node/edge labels and node/edge diagramming symbols (icons) as well as the structure of the graph. Additionally, the generation of node/edge labels and diagramming symbols has been made independent of the generation of the graph structure.

2.2 SUMMARY OF ANALYSIS RESULTS

The formalism of a generalized graph generative system has proven to be an effective structure for analyzing graphs and graph-based languages. A major benefit of generalized graph generative systems proved to be their independence from the specific representational form of the graph (e.g., set theoretic or multidimensional) and the method of implementing the graph generative system. Within the formalism of a generalized graph generative system, the capability for generating a set of sets of relations was developed. In this way, the formal syntax of a graph can be augmented to capture additional details in either a one or multidimensional representation. For example, the relations can be used to specify the geometric positions of nodes and edges, hyperedges, and hierarchical graphs. When formalized graph-based languages are applied to specific problems, the value of this extension was clearly demonstrated. The formalism of an extended graph grammar also proved to be an effective structure for implementing graph generative systems. The separation of the graph structure generation from the label and diagramming symbol generation is an important characteristic. Independence of the generation process is thereby achieved which simplifies many of the analysis questions and provides a great deal of flexibility in choosing the generating functions associated with the extended graph grammar.

Most importantly, the concepts of generalized graph generative systems and generalized graph grammars provided a mathematical vehicle for formally analyzing graphs and graph-based languages. Some of the more important analysis results are summarized below. A detailed treatment of these results is contained in [14].

- Set-Theoretic Analysis: The base sets associated with formalized graphs and graph-based languages are fundamentally different than the ones associated with one dimensional formal languages. For example, the base set $\mathcal{G}^*$ of all graphs over a label alphabet Δ and a diagramming symbol alphabet Σ was shown to be uncountably infinite and not Turing/Recursively enumerable. Therefore, this base set must be partitioned to find a base set(s) that is countable and Turing/Recursively enumerable. The set of all graphs with finite node, edge, and relationship sets was found to be such a set, and this set, denoted $\mathcal{G}_F^*$, is the base set for all formal, graph-based languages.
- Formal Language Analysis: A hierarchy of graphs and graph-based languages was established. Additionally, by choosing one of the several possible representations for graphs as a "standard" representation, a unified hierarchy was developed.
- Algebraic Analysis: The classes of graphs and graph-based languages established by the above hierarchy are closed under the algebraic operations of union and substitution. Additionally, it was shown that algebraic operations can be used to combine independent languages to form a "customized" graph-based language. While it is not always an easy task to develop a grammar for such a language combination, it was shown that the concept of an extended graph grammar can simplify the task.
- Automata Theory Analysis: Turing machine based recognizers for graphs and graph-based languages exist, and they precisely recognize/accept the class of graph-based languages produced by the generalized graph generative systems. Additionally, these machines can be used to establish a complexity hierarchy for the languages.
- Generating Function Analysis: Meta-level generating functions exist which can generate the class of graphs $\mathcal{G}_F^*$ over given alphabets Δ and Σ. Additionally, these functions can be easily adapted to generate specialized subsets of $\mathcal{G}_F^*$, and they serve as a framework for studying the problem of constructing effective and efficient parsers for graphs and graph-based languages.

3 APPLICATION OF THE MODEL

As stated in Section 1, the application approach is to use a formalized, graph-based language as a front-end to a wide-spectrum language. The benefit of using a graph-based language as the initial specification language is in its ability to visually display the complex relationships associated with parallel software. For example, a coarse grain specification and design for parallel software involves communication path relationships, spatial relationships, hierarchical relationships, precedence relationships, and temporal relationships. While these relationships are difficult to conceptualize and visualize with a one dimensional language, they are easy to conceptualize and visualize with

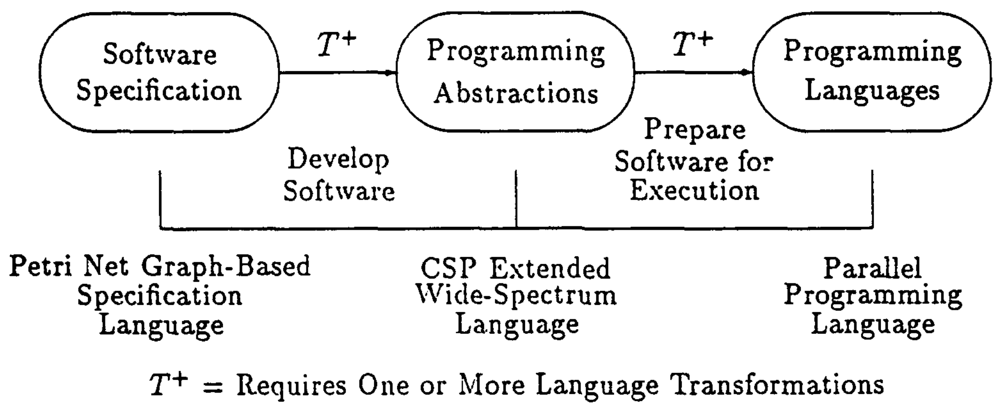

Software Specification
T^+
Programming Abstractions
T^+
Programming Languages
Develop Software
Prepare Software for Execution
Petri Net Graph-Based Specification Language
CSP Extended Wide-Spectrum Language
Parallel Programming Language
T^+ = Requires One or More Language Transformations

Figure 2: Example Framework for Developing Parallel Software

a graph-based language. For example, the nodes of a graph can be used to represent the parallel processes and the edges can be used to represent the communication paths. Additionally, graph relations such as hyperedges can be used to represent the spatial distribution of the processes on a set of parallel processors. Also, by using a more complex node labeling scheme, it is possible to associate attributes and values with the nodes and edges to capture other types of desired information.

To demonstrate the feasibility of the application framework, the Petri net model of parallel computation was chosen as the interface into a wide-spectrum language. No specific wide-spectrum language was chosen for the framework. Instead, the Communicating Sequential Processes (CSP) algebraic model of parallel computation was chosen to extend wide-spectrum languages for the representation of parallelism. The example framework to be analyzed is illustrated in Figure 2. Even though a specific wide-spectrum language was not chosen, it is anticipated that wide-spectrum languages such as REFINE^{TM} [15] are viable candidates for the example framework. Note that the selection of Petri nets and CSP in no way implies that these are the only models that could be used. The intent of choosing the Petri net and CSP models was to perform a detailed analysis of the feasibility of the example framework. Other models could have been chosen as well.

In terms of the front-end graph-based language, several graph-based language models of parallel computation exist such as [16]: computation graphs, finite state machines, message passing systems, parallel program schemata, and Petri nets. However, Petri nets were selected as the initial candidate for the following reasons. Petri nets are a widely used model of parallel computation whose syntax can be formalized as both a set theoretic and a graph-based language. Additionally, Petri nets can be formally analyzed to determine the properties associated with a Petri net specification of a parallel system [16]. In terms of modeling power, Petri nets have been shown to be one of the more expressive models of parallel computation in terms of the types of concurrent/parallel systems they can represent [16]. For example, any parallel computation that can be modeled by a computation graph, finite state machine, or a message passing system can be modeled by a Petri net. However, the converse is not always true.

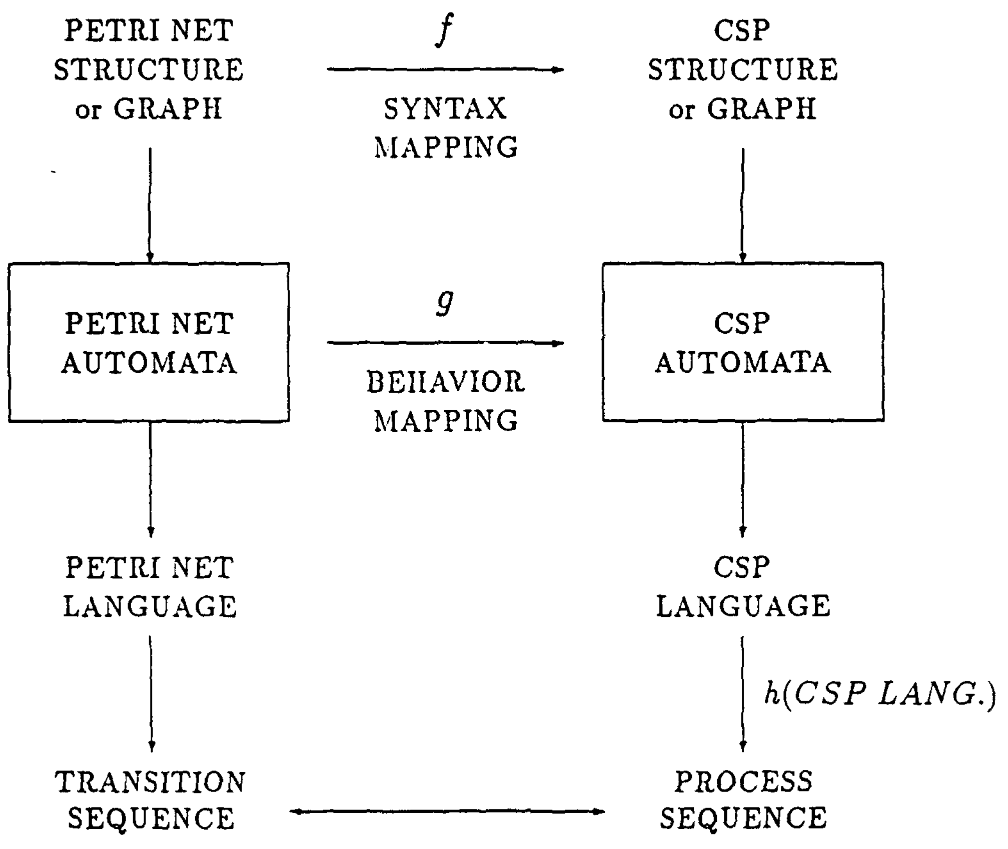
In terms of the wide-spectrum language extension for representing parallelism, several possibilities also existed. Some examples are temporal logic, the algebraic models of communicating processes, actors, interval calculus and UNITY. However, the algebraic model of Communicating Sequential Processes (CSP) was selected as the initial model for the following reasons. It is a useful model for the specification, design, and implementation of computer systems which continuously act and interact with their environment [17]. Many of the systems that exhibit a potential for parallel behavior are systems such as these. Additionally, CSP has been used as the basis for developing several experimental programming environments for parallel software [18, 19, 20]. CSP has been used as the model for developing several parallel programming languages such as Ada and Occam [17]. CSP includes a formal semantic system for analyzing the behavior of parallel software specified with CSP syntax [17]. CSP is based on events, and it is possible to hierarchically decompose the events as the specification and design process proceeds from the initial specification to the final design of the parallel software [17]. Lastly, simulators can be constructed for the CSP model which means CSP specifications are executable and provide a rapid prototyping capability.

In addition to the reasons why Petri nets and CSP were selected, the limitations of these languages need to be stated as well. First, both Petri nets and CSP are static models of parallel computation. That is, the set of transitions of a Petri net and the set of processes in a CSP system are fixed. Additionally, both Petri nets and CSP abstract timing considerations. For example, a Petri net transition executes in zero time and the passing of tokens from a place to a transition and vice versa takes zero time. Similarly, CSP events have no duration, and the time associated with the execution of a process is not considered to be important. While this type of implementation detail is rightfully abstracted ıt the initial specification level, at some point in the further specification, design, and implementation of parallel software, timing considerations are important and must be dealt with.

3.1 ANALYSIS OF THE APPLICATION FRAMEWORK.

The objective of this section is to show that under a syntactic mapping function f_s the behavior of a Petri net graph and its image (a CSP structure) can be compared and that under certain conditions the behaviors are equivalent. It is important to note that many different types of Petri nets have been defined; however, this research concentrated on the class of Petri nets defined in Chapter Two of Peterson [16]. This class is known as the class of general Petri nets. All other classes of Petri nets are specialized or constrained variants of this class. Thus, it makes sense to examine this class first since the results of the analysis app'v to all the other classes as well. Syntactically, this class can be constrained to create the classes of c 'inary Petri nets (no multiple communication links), self loop free Petri nets (no place is both the an input and an output of a transition), and restricted Petri nets (no multiple communication links or loops) [16]. Additionally, the behavior rules of general Petri nets can be modified to create Timed Petri nets, Stochastic Petri nets, and others.

The analysis of the proposed application framework requires a substantial amount of work. In particular, the Petri net to CSP mappings must be examined from both a syntactical and a behavioral perspective. From a syntactical perspective, the mappings must be analyzed to ensure no information is lost when mapping from the Petri net model to the CSP model. From a behavioral perspective, if the Petri net model possesses certain behavioral properties, such as absence of



Fig·ıre 3: Analysis Framework

deadlock, these properties must be preserved under the mapping. Insufficient space exists within this paper to describe the method used to demonstrate the syntactical and behavioral properties of mapping Petri nets to CSP structures. Therefore, only a summary of the basic approach used to analyze the syntactical and behavioral properties under the mappings is provided, and the reader is referred to [14] for additional details. The framework used for the analysis is shown in Figure 3.

Conceptually, the analysis framework can be described as follows. Given a specific instance of a well-formed Petri net graph, the syntax mapping f is used to obtain a corresponding CSP structure. Using an automata theory approach, both Petri net and CSP automata can be defined. If a behavior mapping function g directly maps the behavior of the Petri net automata one-to-one and onto the behavior of a CSP automata, the behavior of the two automata are equivalent. That is, g constrains the operation of the CSP automata to mirror that of the Petri net automata. Thereby, the syntax mapping f and the behavior mapping g can be used to construct an analysis framework that allows for the partial solvability of the behavioral equivalence problem. Additionally, the automata can be programmed to output the execution sequence. Based on this sequence, a selection function h can be defined which converts a finite event sequence generated by a CSP automata into a finite process sequence. Thus, the function h allows for a Petri net transition sequence (or a finite set of sequences) generated by a Petri net automata to be compared with a process sequence (or a finite set of sequences) derived from a CSP event sequence generated by a CSP automata. An additional benefit of this framework is that it allows for the construction of a set of first-order predicate calculus well-formed formulas that can be used to check for the preservation of safety and liveness properties under the mappings f and g.

Using the analysis framework of Figure 3, the following properties were shown:

- For an arbitrary, well-formed Petri net graph, there exists a function f that maps the Petri net syntax one-to-one and onto a corresponding CSP structure. However, the CSP structure is not complete in the sense that the Petri net graph contains insufficient syntactical information to construct the CSP process alphabet and the process expression for the overall CSP process. These must ge generated by alternate means.
- Given the existence of a syntactical mapping function f, there exists a behavior mapping function g that produces equivalent behavior in the Petri net and CSP automata. However, this is true if and only if the function g is direct, one-to-one, and onto.
- Given the syntactical mapping f and the behavior mapping g, there exist first-order predicate calculus well-formed formulas (wffs) that can be used to prove preservation of safety and liveness properties under the mappings. An additional benefit of the wffs is that they allow the determination of whether the definition of a program property is the same (consistent) between two (or more) different models of parallel computation.

4 CONCLUSIONS

As a result of this effort, a formalism for graph-based languages exists that is an effective basis for the specification, analysis, and mapping of graph-based languages. Additionally, the formalism and its associated theoretical foundation provide a solid basis upon which to build many types of application frameworks for the developed theory. Using the theoretical foundation, the formal language theory model of graphs and graph-based languages was connected with the specification and design of parallel software. To determine the feasibililty of this application, the formalized, graph-based syntax of the Petri net model of parallel computation was mapped to the corresponding linear syntax of the Communicating Sequential Processes (CSP) model of parallel computation. CSP is used as the formalism for extending a wide-spectrum language to include explicit constructs for parallel processing, and an applications framework linking the developed theory to wide-spectrum languages such as REFINE^{TM} is suggested. Such a link should provide an easy to use yet formal and disciplined approach to the development of parallel software. Also, the feasibility analysis demonstrated that the behavior of a CSP system obtained via a syntax mapping from a Petri net can be made to be equivalent to the behavior of the Petri net.

References

[1] M. Nagl, "A Tutorial and Bibliographical Survey on Graph Grammars," in *Proceedings of the International Workshop on Graph Grammars and Their Application to Computer Science and Biology (Lecture Notes in Computer Science: 73)*, pp. 70–126, Berlin: Springer-Verlag, 1979.

[2] D. Gelernter, "Domesticating Parallelism," *IEEE Computer*, vol. 19, no. 8, pp. 12–16, August 1986.

[3] S. K. Tripathi *et al.*, "Report on The Workshop On Design and Performance Issues in Parallel Architectures," *ACM SIGMETRICS Performance Evaluation Review*, vol. 14, no. 3 and 4, pp. 16–29, January 1987.

[4] L. H. Jamieson *et al.*, *The Characteristics of Parallel Algorithms.* Cambridge, MA: The MIT Press, 1987.

[5] Z. Manna and P. Wolper, "Synthesis of Communicating Processes from Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 1, pp. 68–93, January 1984.

[6] P. Ladkin, "Specification of Time Dependencies and Synthesis of Concurrent Processes," in *Proceedings of the International Conference on Software Engineering*, pp. 106–115, New York: ACM Press, 1987.

[7] C. L. Group, *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L (Lecture Notes in Computer Science: 183).* Berlin: Springer-Verlag, 1985.

[8] D. C. Luckham *et al.*, "An Environment for Ada Software Development Based on Formal Specification," *Ada Letters*, vol. VII, no. 3, pp. 94–106, May-June 1987.

[9] J. L. Pfaltz and A. Rosenfeld, "Web Grammars," in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 609–619, 1969.

[10] T. W. Pratt, "Pair Grammars, Graph Languages and String-to-Graph Translations," *Journal of Computer and System Sciences*, vol. 5, no. 6, pp. 560–595, December 1971.

[11] P. A. Ng and S. Y. Bang, "Toward a Mathematical Theory of Graph-Generative Systems and its Applications," *Information Sciences*, vol. 11, no. 3, pp. 223–250, 1976.

[12] M. Nagl, "Formal Languages of Labelled graphs," *Computing*, vol. 16, no. 1-2, pp. 113–137, 1976.

[13] H. Ehrig, "Introduction to the Algebraic Theory of Graph Grammars," in *Proceedings of the International Workshop on Graph Grammars and Their Application to Computer Science and Biology (Lecture Notes in Computer Science: 73)*, pp. 1–69, Berlin: Springer-Verlag, 1979.

[14] P. D. Bailor, *A Theory for Graph-Based Language Specification, Analysis, and Mapping with Application to the Development of Parallel Software.* PhD thesis, School of Engineering, Air Force Institute of Technology, (AU), Wright-Patterson AFB, OH., September 1989.

[15] Reasoning Systems Inc., Palo Alto, California, *REFINE Language Summary and Data Sheet*, 1987.

[16] J. L. Peterson, *Petri Net Theory and the Modeling of Systems.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

[17] C. A. R. Hoare, *Communicating Sequential Processes.* London: Prentice-Hall International, 1985.

[18] N. Delisle and M. Schwartz, "A Programming Environment for CSP," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 34–41, New York: ACM press, 1987.

[19] N. D. Francesco and G. Vaglini, "Description of a Tool for Specifying and Prototyping Concurrent Programs," *IEEE Transactions on Software Engineering*, vol. 14, no. 11, pp. 1554–1564, November 1988.

[20] M.-C. Pong, "A Graphical Language for Concurrent Programming," in *Proceedings of the IEEE Computer Society Workshop on Visual Languages*, pp. 26–33, New York: IEEE Press, 1986.

Text Layout As A Problem of Modality Selection

Yigal Arens and Eduard H. Hovy[*]
Information Sciences Institute
The University of Southern California
Marin~ del Rey, CA 90292-6695
Tel: 213-822-1511
ARENS@ISI.EDU, HOVY@ISI.EDU

ABSTRACT

It is not enough for natural language text simply to be generated, it must also be layed out in an appropriate format. Different modes of text (plain text, itemized lists, enumerations, inserts, etc.) are used systematically in order to convey information additional to the primary content of the text. In this paper we apply and extend ideas from earlier work on the automatic allocation of presentation media to given information, and on text planning, to the problem of text layout.

1 Introduction: The Problem of Laying Out Text

The problem of natural language text generation is being, and has been, addressed by many. Typically, researchers choose to concentrate on the problem of generating single sentence, or possibly paragraphs. However, recalling the printed texts we see around us everyday, it is clear that the task of creating natural language text does not end with outputting a paragraph in a natural language. Ultimately, a text generator must decide not just *what* to say, but also how to *organize* the text, and how to *shape* it. For this purpose we have at our disposal numerous **Textual Devices** — itemized lists, indented paragraphs, enumerations, sidebars, footnotes, italicization, quotation, and more. This paper attempts to provide some insight into how the text layout problem can be handled.

Our approach to this problem borrows from ideas and techniques proven useful in the following areas, which we have been studying at ISI for the past several years:

- The use of planning in the generation of natural language text; and

[*]This author was supported in part by the Rome Air Development Center under RADC contract FQ7619-89-03326-0001.

- The automatic selection of appropriate modalities for multi-modal presentation of information.

The next two sections describe the techniques of each of these two problems, and how they have been extended to handle the requirements of text layout. The last section provides an example of how our approach is applied to the generation of a recipe — a type of document which would be rather difficult to comprehend without appropriately used complex textual devices.

2 Automatic Modality Selection

One of our principal themes of research has been the communication of computer-internal information through multiple modalities, including natural language text, tables, maps, graphs, pictures, menus, and icons. One central question of current interest is how to automatically determine the appropriate modality when given a collection of information to present.

In practice, this question can only be settled by a set of heuristic strategies and preferences. The strategies will provide the ability to reason about the natures of various modalities and of various types of data and to find some acceptable (if not the the best possible!) fit, taking into account the current display, the costs of various modalities' activations, the user's preferences, and so forth.

We have recently started work on articulating the precise nature of modalities and of data, and also of developing a terminology of features of both (see [Hovy & Arens 90, Arens & Hovy 90]), and we have rough ideas about the type of planning/negotiation scheme that could perform the task of data-to-modality (or modalities) allocation.

To illustrate the flavor of our work in this area, we provide a partial characterization of modalities we handle in Table 1. Modality selection rules select modalities for given data by identifying modality charactersitics most suitable for displaying the data, and choosing a modality which combines those charactersitics. Consult the work cited above for further details.

In the course of our work on this topic, we noticed an interesting fact: the different text layouts and styles (plain text, itemized lists, enumerations, italicized text, inserts, etc.) — which we refer to as **Textual Devices** — *are used systematically in order to convey information*. The information they convey supplements the primary content of the text. The systematicity holds across various types of texts, genres, and registers of formality. It is found in books, articles, papers, letters, and even notes.

We now believe that one can treat the different textual devices as different modalities. That is to say, the same type of reasoning that goes into the central data-to-modality allocation problem, namely deciding when and how to choose between using a picture or a table or a sentence, should go into deciding whether to generate a straight paragraph or to use an enumerated list or a table or an insert. The reasoning is based, naturally, on the contents to be

Generic Modality	Carrier Dimension	Int. Semantic Dim.	Temporal Endurance	Granularity	Medium Type	Default Detectability	Baggage
Beep	0D		transient	N/A	aural	high	
Icon	0D		permanent	N/A	visual	low	
Map	2D	2D	permanent	continuous	visual	low	high
Picture	2D	3D	permanent	continuous	visual	low	high
Table	2D	2D	permanent	discrete	visual	low	high
Form	2D	>2D	permanent	discrete	visual	low	high
Graph	2D	2D	permanent	continuous	visual	low	high
Ordered list	1D	#D	permanent	discrete	visual	low	low
Unordered list	0D	#D	permanent	N/A	visual	low	low
Written sentence	1D	∞D	permanent	discrete	visual	low	low
Spoken sentence	1D	∞D	transient	discrete	aural	medhigh	low
Animated material	2D	3D	transient	continuous	visual	high	high
Music	1D	?	transient	continuous	aural	med	low

Table 1: Modality characteristics.

conveyed to the reader. The process of device selection, like that of modality selection in general, consists of choosing one (or more) devices whose characteristics are suited for conveniently expressing essential portions of the contents.

2.1 A Characterization of Textual Devices

As is the case with modalities in general, we find that textual devices are distinguished along several independent dimensions. In this case: *Variation*, *Position*, and *Composition*.

Variation:

The method by, and extent to which, a textual device involves modification or supplementing the actual text string used. Examples of devices which are primarily variational: parenthesization, font switching (e.g., italicization *when the surrounding text is* not *italicized*), and capitalization.

Text variation is used to express a property of the varied text, with the precise property depending on the type of variation. For example, parentheses indicate that the parenthesized text is tangential to the main text. Font switching indicates special importance of the text, that it expresses a new term being introduced for the first time, or that it is in a foreign language. Capitalization indicates that the text string names a particular entity. Quotation marks are usually used, as their name makes clear, to indicate that the text they enclose is a direct quote of someone other than the author of the surrounding text. They can also be used to indicate that the *meaning* of the quoted text is different than a standard interpretation would yield.

Position:

The extent to which a textual device involves moving the text string in relation to the surrounding text or the page. We have identified three primary values of position: Inline (non-distinguished), Offset (as with an indented paragraph, or a long quotation), and Offpage (e.g., a footnote, or a sidebar).

Shifting of a text block's position is used to indicate the relationship of the affected text to the surrounding text. Text is offset to indicate that it is authored by someone else (e.g., a long quoted paragraph), or that it summarizes a point that is especially relevant. Text is moved offpage to summarize a point that is tangential to the main text, but of explanatory importance.

Composition:

The internal structuring that a device provides the text. Some devices which are primarily compositional are discussed further below.

The semantic contribution of the composition includes device-specific relationships among component subtexts. A large number of compositional devices are in common usage. Several examples:

Itemized list: A set of entities/discourse objects on the same level of specificity with respect to the domain, but with too much material to be expressed about each to allow simple listing.

Enumerated list: A set of entities/discourse objects on the same level of specificity with respect to the domain, which are, furthermore, ordered along some underlying dimension: time, distance, importance, etc.

Elaboration:[1] A pair of texts separated by a colon, where the first is the name of a discourse object and the second defines it, or expresses some other fact related to it. Elaborations are often used within itemized lists.

As we see from the descriptions above, selecting an appropriate textual device (or combination of such) relies on the ability to accurately describe of the semantic content to be expressed by the language and its relationship to the surrounding text. Such an analysis is provided, in part, by text planning research.

[1]This expression refers to a construction of the form "Term: Text string."

3 Text Planning

There is more to building coherent text than the mere generation of single sentences. In order to produce coherent paragraphs (when writing as well as by computer), one requires an understanding of the interrelationships between the parts of a paragraph. For example, the following paragraph is simply not coherent, because the logical interrelationships between the sentences are not respected rhetorically:

> The logical interrelationships between the sentences are not respected rhetorically. One requires an understanding of the interrelationships between the parts of a paragraph. There is more to building coherent paragraphs than the mere generation of single sentences. This paragraph is simply not coherent. One produces coherent paragraphs (when writing as well as by computer).

The question "What makes text coherent?" has a long history, going back at least to [Aristotle 54]. A number of researchers have recognized that, in coherent text, successive pieces of text are related in particular ways, and have provided different sets of interclause relations (see, for example, [Hobbs 79, Grimes 75, Shepherd 26, Reichman 78]). After an extensive study of hundreds of texts of different types and genres, [Mann & Thompson 86, Mann & Thompson 88] identified 25 basic rhetorical relations, which they claimed suffice to represent all intersentential relations that hold within normal English texts. Their theory, called Rhetorical Structure Theory (RST), holds that the relations are used recursively, relating ever smaller blocks of adjacent text, down to the single clause level; it assumes that a paragraph is only coherent if all its parts can eventually be made to fit under one overarching relation. Thus each coherent paragraph can be described by a tree structure that captures the rhetorical dependencies between adjacent clauses and blocks of clauses. Most relations have a characteristic cue word or phrase which informs the hearer or reader how to relate the adjacent parts; for example SEQUENCE is signaled by "then" or "next" and PURPOSE by "in order to". The RST relations subsume most of the rhetorical relations of previous researchers.

Within the past few years, a number of computational research projects have addressed problems that involve generating coherent multisentence paragraphs. Almost all of these use a tree of some kind to represent the structure of the paragraph. An ongoing effort at ISI, headed by one of the authors, uses RST relations (and extensions of them), represented and formalized as plans, in a top-down hierarchical planning system reminiscent of the Artificial Intelligence planning system NOAH [Sacerdoti 77]. The structure planner mediates between some application program (such as an expert system) and the sentence generator Penman [Penman 89, Mann & Matthiessen 83]. From the application system, the planner accepts one or more communicative goals along with a set of clause-sized input entities that represent the material to be generated. During the planning process, it assembles the input entities into a tree that embodies the paragraph structure, in which nonterminals are RST relations and terminal nodes contain the input material. It then traverses the tree, noting the linking phrases at tree branches and submitting the leaves to Penman to be generated in English. The planning process is described in much more detail in [Hovy 88a, Hovy 88b] and in [Moore & Paris 89, Moore & Swartout 89].

4 An Example of Layout Planning

Following is an example of how the text structure planning techniques described above can fruitfully be applied to the selection of appropriate textual devices. We have chosen to speak of the application of a textual device as the execution of a *plan*, to conform to the conventions of text planning.

Consider generating the text for a recipe. The input data consists of ingredients and steps to perform. The data base contains little internal structure beyond the indication of type of information and the temporal sequence of the steps. This, however, is enough to allow the generalized text planner to proceed, given the goal to generate the recipe by starting with the ingredients. If the planner has access to text plans corresponding to the types of textual devices described in Section 2.1, it will be able to recognize that the ENUMERATE plan is not appropriate (since the ingredients, though all on the same hierarchic level, are not ordered), but that the ITEMIZE plan is. It will then collect all the ingredients and itemize them. For the next step, the ENUMERATE plan *is* appropriate, given the underlying temporal dimension ordering the steps, and an enumerated list, headed by *Step N*, will be generated. Since each item consists of an explanation of what is to be done at that step, ELABORATE will be used to provide it structure. If, to make things one level more interesting, some actions are performed simultaneously, then they are not ordered with respect to each other, and the enumerated item will contain them all as straight text. The result will thus be a well-displayed text along the following lines:

```
Ingredients:
  - salt,
  - 2 oz butter,
  - 4 eggs,
  ...

Procedure:
  Step 1: Grind the lemon rind.
  Step 2: Separate the whites from the yolks of the eggs.
  Step 3: Beat the whites until firm, and add the sugar and
          salt. Also add the lemon rind.
  ...
```

A text generator without the layout planning capabilities would produce a straightforward paragraph, much harder to comprehend, along the following lines:

```
  The ingredients are salt, 2 oz butter, 4 eggs, .... The
  procedure is to first grind the lemon rind, then separate
```

```
the whites from the yolks of the eggs, then to beat the
whites until firm, and add the sugar and salt. Also add
the lemon rind. ....
```

One can imagine how these planning methods can be generalized in order to generate car or airplane repair manuals from computer-internal storage. In fact, it is possible to have the computer generate the appropriate data piece by piece, as it is informed to continue, and even to link such generation to speech synthesis technology. The differences between the speech modality and textual modalities will have to be taken into account in the presentation planning process. In the long term, we expect our approach can be extended to the design of displays incorporating non-textual modalities as well, such as pictures and charts.

References

[Arens & Hovy 90] Arens Y. and E. H. Hovy. How to Describe What? Towards a Theory of Modality Utilization. Submitted to the Twelfth Annual Conference of the Cognitive Science Society, to be held in Boston, Massachusetts, July 1990.

[Aristotle 54] Aristotle. The Rhetoric. In *The rhetoric and the poetics of Aristotle*, W. Rhys Roberts (trans), Random House, New York, 1954.

[Grimes 75] Grimes, J.E. *The thread of discourse*. Mouton, The Hague, 1975.

[Hobbs 79] Hobbs, J.R. Coherence and coreference. *Cognitive Science* 3(1), 1979 (67-90).

[Hovy 88a] Hovy, E.H. Planning coherent multisentential text. In *Proceedings of the 26th ACL Conference*, Buffalo, 1988 (163-169).

[Hovy 88b] Hovy, E.H. Approaches to the planning of coherent text. Presented at the *4th International Workshop on Text Generation*, Los Angeles, 1988. In Paris, C.L., Swartout, W.R. and Mann, W.C. (eds), *Natural Language in Artificial Intelligence and Computational Linguistics*, to appear.

[Hovy & Arens 90] Hovy, E. H. and Y. Arens. Allocating Modalities In Multimedia Communication, in *Working Notes: AAAI Spring Symposium on Knowledge-Based Human-Computer Communication*, Stanford University, California, March 27–29, 1990.

[Mann & Matthiessen 83] Mann, W.C. and Matthiessen, C.M.I.M. Nigel: A systemic grammar for text generation. USC/Information Sciences Institute Research Report RR-83-105, 1983.

[Mann & Thompson 86] Mann, W.C. & Thompson, S.A. Rhetorical Structure Theory: Description and Construction of Text Structures. In *Natural Language Generation: New Results in Artificial Intelligence, Psychology, and Linguistics*, Kempen, G. (ed), Kluwer Academic Publishers, Dordrecht, Boston, 1986 (279-300).

[Mann & Thompson 88] Mann, W.C. and Thompson, S.A. Rhetorical structure theory: Toward a functional theory of text organization. *Text* 8(3), 1988 (243-281). Also available as USC/Information Sciences Institute Research Report RR-87-190.

[Moore & Paris 89] Moore, J.D. and Paris, C.L. Planning Text for Advisory Dialogues. In *Proceedings of the 27th ACL Conference*, Vancouver, 1989.

[Moore & Swartout 89] Moore, J.D. and Swartout, W.R. A reactive approach to explanation. In *Proceedings of the International Joint Conference on Artificial Intelligence IJCAI*, Detroit, 1989.

[Penman 89] *The Penman Documentation*. Unpublished documentation for the Penman language generation system, USC/Information Sciences Institute.

[Reichman 78] Reichman, R. Conversational coherency. *Cognitive Science* 2, 1978 (283-327).

[Sacerdoti 77] Sacerdoti, E. *A structure for plans and behavior*. North-Holland Publishing Company, Amsterdam, 1975.

[Shepherd 26] Shepherd, H.R. *The fine art of writing*. The Macmillan Co, New York, 1926.

KAPTUR: KNOWLEDGE ACQUISITION FOR PRESERVATION OF TRADEOFFS AND UNDERLYING RATIONALES[1]

Sidney C. Bailin
J. Mike Moore
Richard Bentz
Manju Bewtra

CTA INCORPORATED
Rockville, MD

Abstract: We describe an environment that supports the evaluation of potentially reusable artifacts throughout the software development process. The goal of KAPTUR is to harness knowledge gained through successive projects in a given domain, in support of new efforts. Only by understanding the decisions that went into past development efforts can developers intelligently reuse existing artifacts.

The fundamental concept in KAPTUR is the *distinctive feature*, which is any feature of an artifact that differs from common or recommended practice, or that represents a significant development decision. KAPTUR employs hypertext techniques to link artifacts according to their similarities and differences, and to link the distinctive features of an artifact to the supporting rationales, associated tradeoffs, and issues underlying the decisions.

An initial prototype of KAPTUR was developed in 1989. We are currently at work on KAPTUR '90, which builds on the initial prototype, adds some functions deliberately omitted in the first phase, and corrects some deficiencies that we discovered through demonstrating the environment. This year we will also take the first steps towards introducing KAPTUR into a production setting in support of NASA's development of ground system software for unmanned scientific missions.

1 Introduction

KAPTUR—a development environment based on Knowledge Acquisition for Preservation of Tradeoffs and Underlying Rationales—is intended to support systematic reuse of knowledge and artifacts throughout the software development lifecycle. The main contribution of KAPTUR is its support for evaluating potentially reusable artifacts, so that the developer can make an intelligent choice between them. KAPTUR is intended to provide as much information as possible, in an easily accessible form, to clarify whether a given artifact is suitable for reuse in a given context.

KAPTUR is intended to preserve knowledge that is required or generated during the development process, but that is often lost because it is *contextual*, i.e., it does not appear directly in the end-products of development. Such knowledge includes issues that were raised

1 The KAPTUR project is funded by NASA's Code O through Goddard Space Flight Center's Code 522.3.

during development, alternatives that were considered, and the reasons for choosing one alternative over others. Contextual information is usually only maintained as a memory in a developer's mind. As time passes, the memories become more vague and individuals become unavailable, and eventually the knowledge is lost. KAPTUR seeks to mitigate this process of attrition by recording and organizing contextual knowledge as it is generated. From the lessons learned through previous efforts, developers can improve their insight into the current problem and its possible solutions.

KAPTUR employs hypertext techniques to support the content-dependent relationships between artifacts and knowledge from past efforts. This approach results in numerous links between artifacts. For example, two artifacts that incorporate a common element (or group of elements) are linked by the element. Two systems whose designs address a common issue are linked by the issue. As the content evolves, so do the relationships; this is the advantage of linking elements in terms of their content rather than a predefined set of relations.

The fundamental concept in KAPTUR is the *distinctive feature*, which is any feature of an artifact that differs from common practice or that represents a significant decision. It is through the distinctive features of potentially reusable artifacts that the developer evaluates the alternatives for reuse. KAPTUR links artifacts that share a distinctive feature, and it links all of these to artifacts representing alternatives to the feature.

2 A Scenario of the Use of KAPTUR

The idea for KAPTUR grew out of a domain analysis of control center software. In the process of comparing software architectures for different mission control center application processors, we found ourselves reverse engineering the rationales for various decisions. These decisions concerned, for example, the inclusion or omission of functions, the grouping of functions, and the levelling of subsystems and components.

This process suggested that a reuse environment should not simply present to the developer a set of alternative architectures that have been used for previous systems: the developer would have no sound basis on which to select one architecture over the others, to merge aspects of several, or to define yet another software architecture. It is, instead, necessary to present the rationales and issues involved in choosing among the alternatives.

Figure 2-1 illustrates how KAPTUR would be used to explore alternative software architectures for a control center application processor. In the center of the diagram there is a knowledge base containing information about the application domain. This includes recommended architecture(s) and information about previously developed systems. In this scenario, the developer has available a set of software requirements, and wants to begin defining an applications processor to meet these requirements. The developer sits down at the KAPTUR workstation and issues a command whose meaning is something like the following:

> I want to develop a control center applications processor. Show me what they look like.

In response, KAPTUR displays the recommended generic architecture (upper right-hand box).

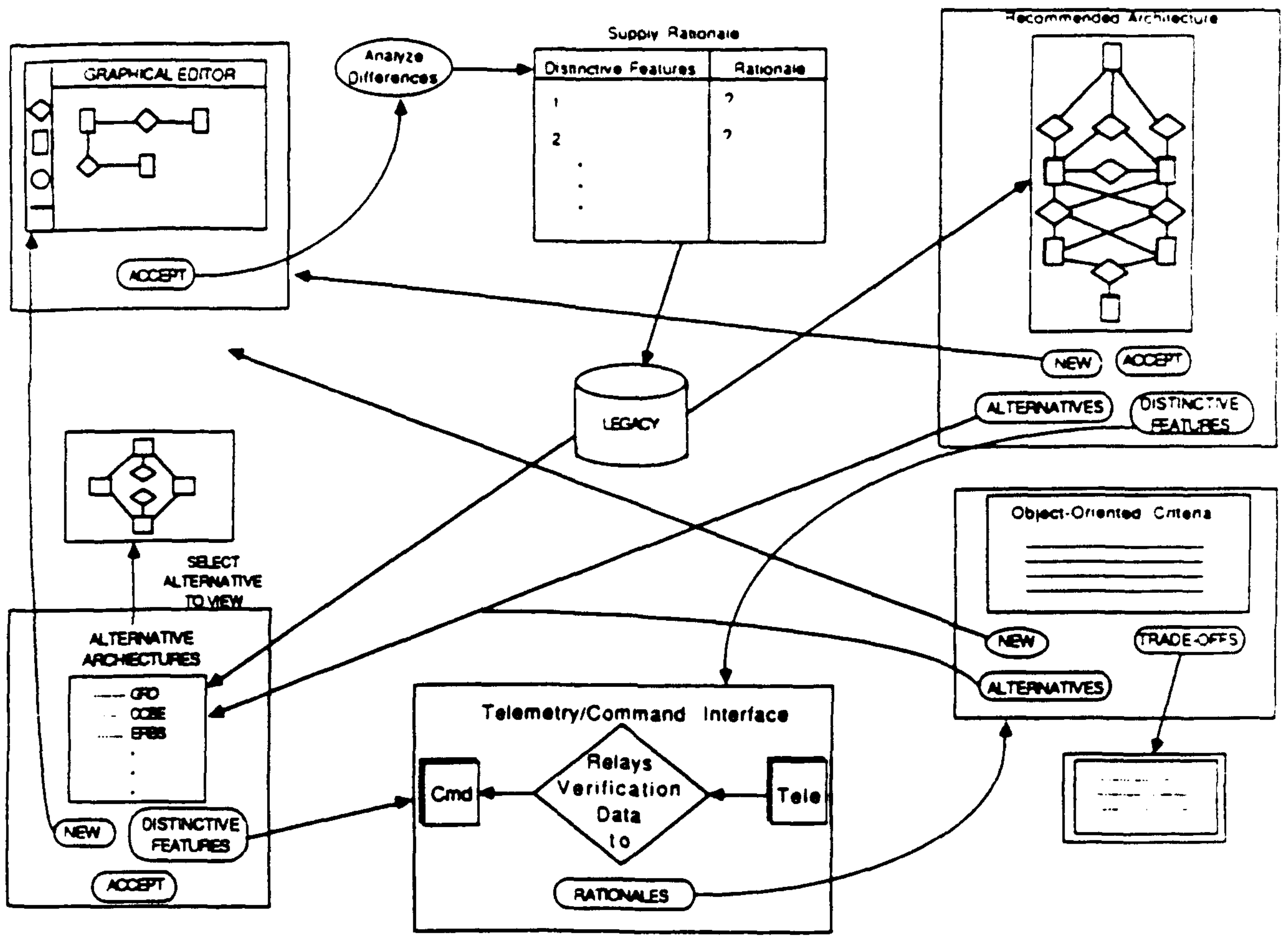


Figure 2-1: KAPTUR may be Used to Explore Alternative Control Center Architectures

In this scenario, there is only one recommended generic architecture because that is the conclusion (to date) of our domain analysis. That conclusion may change as our analysis continues, or as the domain evolves; and certainly at lower levels of the software there will be alternative recommended approaches for different requirements.

Upon examining the recommended architecture, the developer has the following options:

- ACCEPT the recommended architecture
- Examine the DISTINCTIVE FEATURES of the recommended architecture
- Examine ALTERNATIVES to the recommended architecture
- Define a NEW architecture

If the developer selects ALTERNATIVES, KAPTUR displays a list of existing systems that are different from the recommended generic architecture in some significant respect (since the recommended architecture is generic, almost every production system will be different in some respects, if only by instantiating various generic parameters of the recommended architecture). This display is shown in the lower left-hand box of Figure 2-1. The developer can then select one or more of these systems to view their respective architectures. As with the recommended architecture, the developer has the option here of ACCEPTing one of the other architectures, or of deciding to define a NEW architecture.

The DISTINCTIVE FEATURES of an architecture are those that are different from common practice or the recommended approach, or that represent a non-trivial decision about a significant issue. It is the knowledge and analysis underlying these decisions that KAPTUR is intended to preserve. Distinctive features may correspond to specific portions of an architecture (e.g., in Figure 2-1, the interface between the Telemetery and Command Subsystems), or they may represent some aspect of the architecture as a whole (e.g., the distribution of initialization functions to all subsystems).

If the developer selects DISTINCTIVE FEATURES, KAPTUR will list the distinctive features of the architecture being displayed, and will allow the developer to select one or more of these features. KAPTUR will then display a representation of the distinctive feature(s). In effect, the developer has the capability to *zoom* into a view of a particular feature of the architecture. This is illustrated in the bottom-middle box in Figure 2-1.

The developer can then examine the RATIONALES for this feature, i.e., the reasoning underlying the decision that the feature represents. In the lower right-hand box in Figure 2-1, we illustrate the rationales as a list of object-oriented design criteria that might underlie the decision. From this screen, the developer can request even more detailed explanations, by asking to view the TRADE-OFFS that were considered in reaching the decision. The developer can also ask to see ALTERNATIVES to this decision, i.e., other systems that do not possess this feature because a different decision was made.

If the developer selects NEW (from either the Recommended Architecture or the Alternative Architectures screen), a graphical editor will be invoked to allow the interactive definition of the new architecture. However, the definition of the new archtitecture need not proceed completely from scratch. A clipboard capability in KAPTUR will allow the developer to select portions of the recommended and/or alternative architectures for inclusion in the new architecture.

Once a new architecture has been defined, KAPTUR will perform an automated analysis to determine its distinctive features, i.e., the ways in which it is significantly different from the recommended architecture. For each distinctive feature, KAPTUR will prompt the user to enter one or more rationales justifying the feature. This is shown in the top-middle box (above LEGACY) in Figure 2-1.

The new architecture, together with its rationales, then becomes part of the LEGACY of this domain, and will appear in the ALTERNATIVES list when KAPTUR is next used. This is how the evolution of domain requirements and solutions is captured in the knowledge base. At some point, the differences between the recommended architecture(s) and new systems may become so numerous that a reevaluation of the recommended architectures is necessary. The practical need for knowledge base management therefore drives the ongoing domain analysis process, ensuring that domain models are kept current with present requirements.

3 Distinctive Features

Distinctive features are the primary vehicle for comparing alternative reusable artifacts in KAPTUR. The concept of *distinctive feature* is most meaningful when there are norms to which artifacts can be compared. In the absence of such norms, artifacts can be compared with each other; the distinctive features are then the significant ways in which artifacts of the same overall type differ from each other. In any domain with a substantial legacy, however, there will almost certainly be norms (at least implicit ones). Articulating these can help in deciding what constitutes a "significant" difference.

In our domain analysis of control center software, we identified features that fall into in three categories:

- Differences in content
- Differences in leveling
- Differences in grouping

Differences in *content* include variations in functional capability, and dependencies that occur in one artifact but not in another. In the satellite ground system domain, for example, a control center may be responsible for the entire spacecraft or for a specific suite of instruments, resulting in different command scheduling, telemetry validation, and user interface requirements.

Leveling refers to the introduction of aggregate elements, such as objects, subsystems, or high-level processes. Elements previously found at a higher level are now included within these aggregates. In our domain analysis of control center software, we found such aggregates being introduced as the required functionality of the systems became increasingly complex. In the absence of these aggregate, the number of functions at a given level would have grown too large.

Grouping differences pertain to the boundaries of aggregate elements. The distinctive features represent alternative decisions about where to place a given function. For example, in NASA ground systems, the boundary between the Command Management System and the Control Center Applications Processor can vary; certain scheduling functions can be performed in either.

The features just described are all semantically significant, but they are syntactically recognizable as long as there is a means of recognizing equivalent or similar elements in two artifacts. In KAPTUR we rely on similarity of names to make such identifications; other methods might employ keywords or classifications attached to the elements (e.g., functions in the class Telemetry Limit Checking would be recognized as performing similar roles in their respective control centers).

Although the features described above are semantically meaningful, the truly significant differences between two systems might best be described in much higher-level terms, such as the overall design approach taken to meeting perfomance, availability, or security requirements. The current implementation of KAPTUR encodes the syntactic checks as a fixed set of Prolog rules. The development plan for KAPTUR '90, however, includes a language with which the user can specify new rules for detecting distinctive features.

In the long-term concept of operations for KAPTUR, there is another provision for defining new features in terms of lower-level features, which are already known to the environment. When KAPTUR prompts for justification of the features it has detected in a newly entered artifact, the user will be able to group some or all of these features as manifestations of a higher-level feature. In the initial implementation of this capability, KAPTUR will simply identify the higher-level feature with the set of lower-level features. Subsequent implementations may include an explanation-based generalization (EBG) function, which will allow KAPTUR to abstract from the inessential details of the lower-level features, and thus obtain a more robust definition of the higher-level feature. We are currently performing experiments with such an (EBG) function in another (related) task.

4 Knowledge Layers

Knowledge in KAPTUR is stratified into the four layers shown in Figure 4-1: artifacts, similarities and differences, rationales, and underlying issues. Layering the knowledge permits the user to study the KAPTUR database in various degrees of depth. The most superficial view restricts the search to the top (artifact) layer; as the lower layers are viewed, the analysis of alternatives becomes deeper.

The artifact layer represents the repository of reusable products from the entire software lifecycle, e.g., requirements specifications, designs, code modules, test plans and histories, user documentation, etc. We chose the term *artifact* in order to be as general as possible, so that the reusable products are not restricted to those of a specific lifecycle phase, lifecycle model, or development methodology.

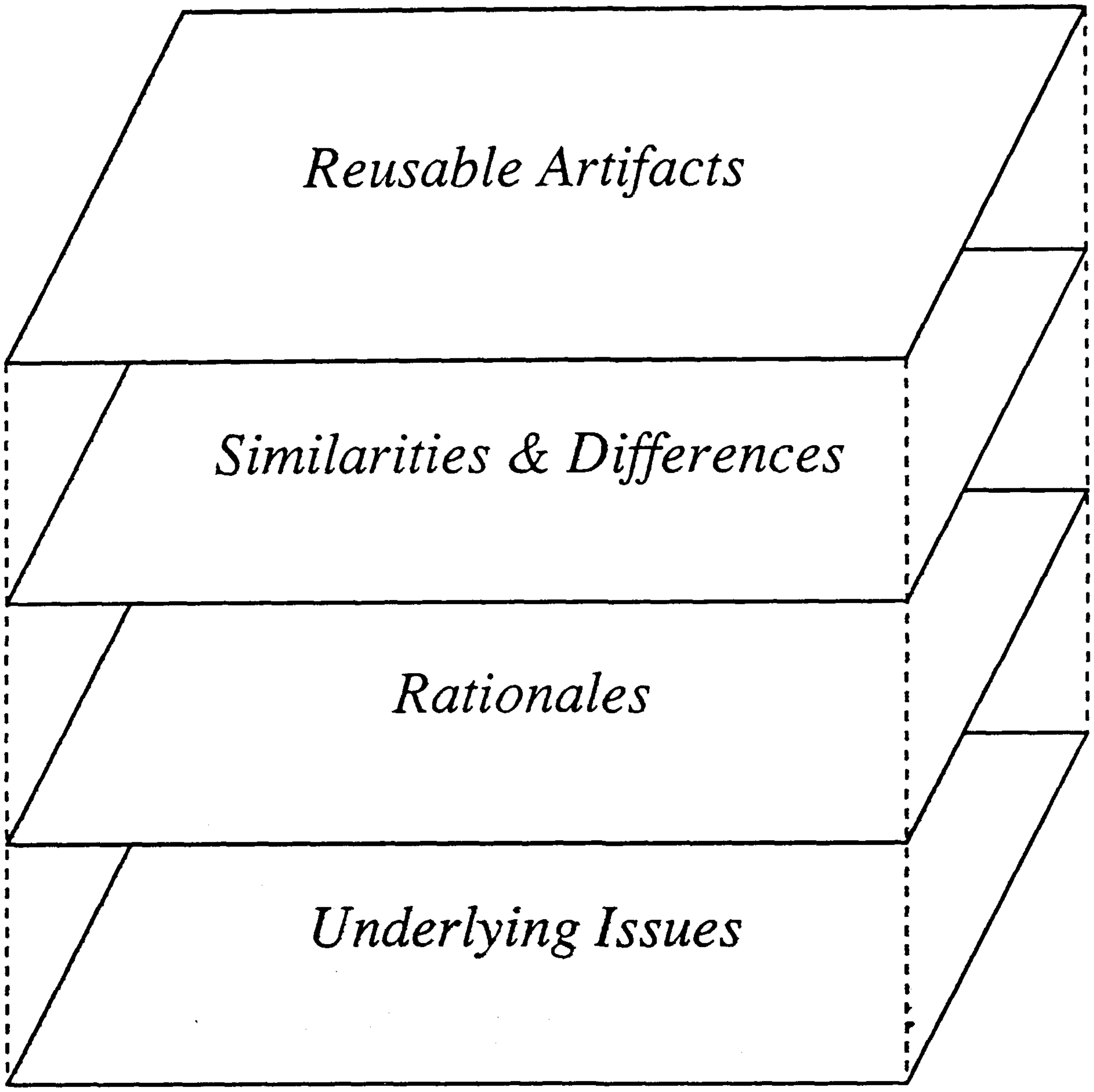


Figure 4-1: Layers in KAPTUR's Knowledge Base

We support this generality by distinguishing between the artifact itself and its description within KAPTUR. The artifact itself is not necessarily found in the KAPTUR database; ordinarily, it will lie elsewhere. This is true even when KAPTUR contains diagrams describing, say, the design of a system or subsystem. Such diagrams are not necessarily those of the system's design specification. They represent a *synopsis* of the information in the design specification, in terms understood by KAPTUR. This approach allows us to choose a uniform set of graphical representations of designs (uniformity is necessary in order to compare alternative designs intelligently) without restricting the database to systems that were designed in the chosen notation.

Below the artifact layer are the similarities and differences between artifacts. Distinctive features appear at this level. The hypertext mechanism serves as a means of linking artifacts that share a distinctive feature, as well as those that represent alternatives. For example, having identified a distinctive feature of a given artifact, the user can look at artifacts of the same type that do *not* possess this feature.

Commonalities in content also appear at this level. If two artifacts have been derived from a single reused template, this fact is represented by virtual links not only between each instance and the template but also between the instances themselves. Such links help mitigate the loss of information through inadequate distinctive feature recognition rules. Implicitly there must be some feature that distinguishes the two instances, but it may not be caught by the rules that have been specified to date (these rules evolve as part of KAPTUR's model of the domain).

Below the similarities and differences are the rationales for the distinctive features. This layer documents the contextual knowledge that often is lost when projects end. The distinctive features are meant to include all significant decisions that went into developing an artifact in a particular way. The rationales provide a history of the reasoning that went into the development. In addition to the rationales, this layer also records any tradeoffs that were considered in reaching the final decisions.

Links at this layer provide further insight into the alternatives for reuse. The user can proceed from a distinctive feature of an artifact to the justification of the feature and the tradeoffs considered. Cases that resolved the tradeoffs differently can then be viewed. Links at this level can reach contexts not available through links at the higher levels. A given tradeoff, such as space vs. time, may arise in contexts altogether different from the original artifacts the user was viewing. The layering allows KAPTUR to make such breadth of information available to the user when needed, without overloading him when it is not required.

The fourth layer describes issues underlying the rationales and tradeoffs. This layer provides the broadest coverage of interrelated knowledge. The user can start from a design principle, such as segregating functions that are likely to change (cited as a rationale for some feature), and link to an underlying issue such as "What are the object-oriented criteria for decomposing a system?" From here, other decomposition criteria may be viewed, such as grouping functions that operate on the same data. The user may then want to gauge the effectiveness of the criteria by linking to artifacts that exhibit them.

Issues themselves are stratified into levels of generality. For example, the issue just cited, object-oriented decomposition criteria, might be linked to the higher-level issue of alternative design and programming methodologies, and the conditions under which they should be used.

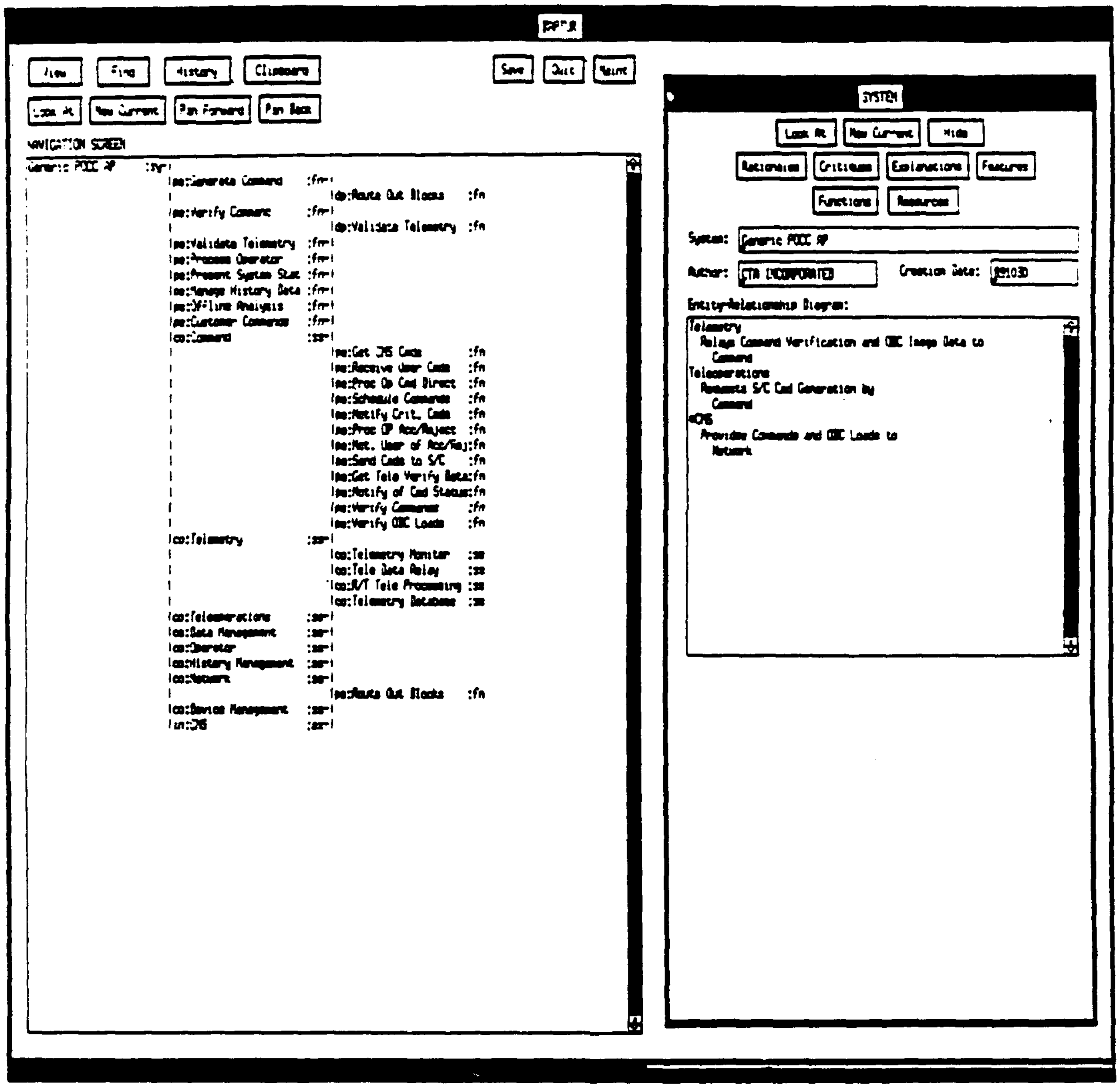
5 Initial Implementation

The first prototype of KAPTUR was delivered to NASA in November, 1989. Figure 5-1 shows the user interface. We made a deliberate decision to defer a graphical user interface until later in order to make progress on the essential capabilities first. A graphical interface has now been implemented for KAPTUR '90.

The interface consists of two primary windows: the Navigation Screen and the Current Node Screen. The Navigation Screen is intended to provide the user with a broad view of the knowledge base contents; these are displayed in neighborhoods consisting of elements directly or indirectly linked to a specified "current" node. The Navigation Screen supports panning back and forth through neighborhoods. The view can be interactively tailored by the user to filter out irrelevant information. Automated search for a node with given characteristics can be activated through a Find command, which is available from the Navigation Screen.

The Current Node Screen provides detail on a selected node. It is here, for example, that the diagrams describing an artifact are presented (the initial implementation provides a textual form of entity-relationship diagrams for describing systems and subsystems). There is also a Lookat Screen, which has the same form as the Current Node Screen, but which is more dynamic: the node being "looked at" changes as links are traversed, while the current node does not change until the user explicitly requests this. A history command presents the user with a record of the previous current nodes, and a record of the nodes traversed in the Lookat Screen; by selecting a node from these lists, the user can return to any past position within the session.

The knowledge base is an implementation of an entity-relationship network structure. To drive the distinctive feature analysis, this structure is translated into a set of Prolog facts; the two representations are maintained automatically in parallel. This approach was chosen for convenience in developing a prototype quickly. As part of the KAPTUR '90 effort, we are evaluating long-term solutions to the knowledge base and distinctive feature analysis requirements.



NAVIGATION SCREEN

```
Generic POCC AP      :sy-|
|                        |pe:Generate Command    :fn-|
|                        |                            |dp:Route Out Blocks     :fn
|                        |pe:Verify Command      :fn-|
|                        |                            |dp:Validate Telemetry   :fn
|                        |pe:Validate Telemetry  :fn-|
|                        |pe:Process Operator    :fn-|
|                        |pe:Present System Stat :fn-|
|                        |pe:Manage History Data :fn-|
|                        |pe:Off-line Analysis   :fn-|
|                        |pe:Customer Commands   :fn-|
|                        |co:Command             :ss-|
|                        |                            |pe:Get CMS Cmds         :fn
|                        |                            |pe:Receive User Cmds    :fn
|                        |                            |pe:Proc Op Cmd Direct   :fn
|                        |                            |pe:Schedule Commands    :fn
|                        |                            |pe:Notify Crit. Cmds    :fn
|                        |                            |pe:Proc OP Acc/Reject   :fn
|                        |                            |pe:Not. User of Acc/Rej:fn
|                        |                            |pe:Send Cmds to S/C     :fn
|                        |                            |pe:Get Tele Verify Data:fn
|                        |                            |pe:Notify of Cmd Status:fn
|                        |                            |pe:Verify Commands      :fn
|                        |                            |pe:Verify OBC Loads     :fn
|                        |co:Telemetry           :ss-|
|                        |                            |co:Telemetry Monitor    :ss
|                        |                            |co:Tele Data Relay      :ss
|                        |                            |co:R/T Tele Processing  :ss
|                        |                            |co:Telemetry Database   :ss
|                        |co:Teleoperations      :ss-|
|                        |co:Data Management     :ss-|
|                        |co:Operator            :ss-|
|                        |co:History Management  :ss-|
|                        |co:Network             :ss-|
|                        |                            |pe:Route Out Blocks     :fn
|                        |co:Device Management   :ss-|
|                        |un:CMS                 :ex-|
```

System: Generic POCC AP

Author: CTA INCORPORATED   Creation Date: 891030

Entity-Relationship Diagram:

```
Telemetry
  Relays Command Verification and OBC Image Data to
    Command
Teleoperations
  Requests S/C Cmd Generation by
    Command
NCMS
  Provides Commands and OBC Loads to
    Network
```

Figure 5-1: The KAPTUR '89 User Interface

UTILIZING INTERACTIVE MULTIMEDIA TO SUPPORT KNOWLEDGE-BASED DEVELOPMENT OF SOFTWARE REQUIREMENTS

James D. Palmer and Peter Aiken

Center for Software Systems Engineering - Mail Stop ST-103

George Mason University, Fairfax, VA 22030-4444

703/323-3891 or 703/323-4299

e-mail: jpalmer@gmuvax or paiken@gmuvax2.gmu.edu

ABSTRACT

One of the major sources of problems associated with requirements engineering processes falls at the interface between user and analyst and is concentrated on information transfer and information transformation. This includes failure to provide a robust environment in support of information transfer and transformation needs. A paradigm based on the utilization of knowledge-based tools to aid in the formulation of multimedia-based software requirements is presented as one approach to address these issues. We have combined these two technologies to create a workstation for application in computer supported group requirements efforts. The workstation is described in terms of architecture, system tool kit, and a number of features offering specific support for group work. We present the initial results of application of the workstation to several sets of requirements. Initial results indicate that this approach is capable of producing useful knowledge-based assistance to users and analysts concerned with developing software requirements.

INTRODUCTION

Requirements engineering for large complex systems is seldom performed by one or two individuals setting about their work with documents and interviews to generate requirements. Rather, it is nearly always based on work performed by groups or teams who have particular assignments and utilize information from any media format that may be available to assist them. This may include documents, interviews, tape recordings, video tapes, viewing the system to be modified, simulation, animation, and various combinations of these. We are developing the multimedia workstation environment in support of this approach to requirements engineering and as a means of providing support for computer supported cooperative work (CSCW).

The workstation concept has evolved from our efforts to resolve information transfer and information transformation difficulties between groups of users, analysts, and other development personnel functioning at the interface between user and analyst groups. Transfer and transformation problems arise between user and analyst groups due to issues such as language imprecision, ambiguous statements, lack of detailed knowledge about the system or development approach, and requirements volatility. This is shown conceptually in Figure 1. Initial efforts were aimed at resolving issues related to system performance as reflected in user selection of quality factors. From this we turned to the development of a tool to analyze, classify, and perform diagnostics on requirements statements. The primary region of concern has been the interface between user and analyst. Since this activity is fundamentally performed by groups, we have initiated the development of a computer supported cooperative work environment that extends the information bandwidth to include all forms of media, not just text and graphics.

Information transfer difficulties occur as users attempt to describe the problem domain to analysts, while information transformation difficulties occur as analysts codify and record requirements information in forms useful for design. We have studied the transfer of requirements information from user to analyst and its succeeding transformation into formats for use in subsequent phases of the software development life cycle and have found that most of the problems occur at the interface between user and analyst during the requirements elicitation, analysis, and specification phase [Sage90]. The most serious of these errors relates to misinterpretation of user requirements, with the result being systems that do not satisfy user needs. As depicted in Figure 1, errors introduced in this initial phase, that remain undetected, propagate through subsequent development phases often requiring significant resources to correct when discovered. Occasionally, failure to discover software errors has had tragic consequences [Subc89].

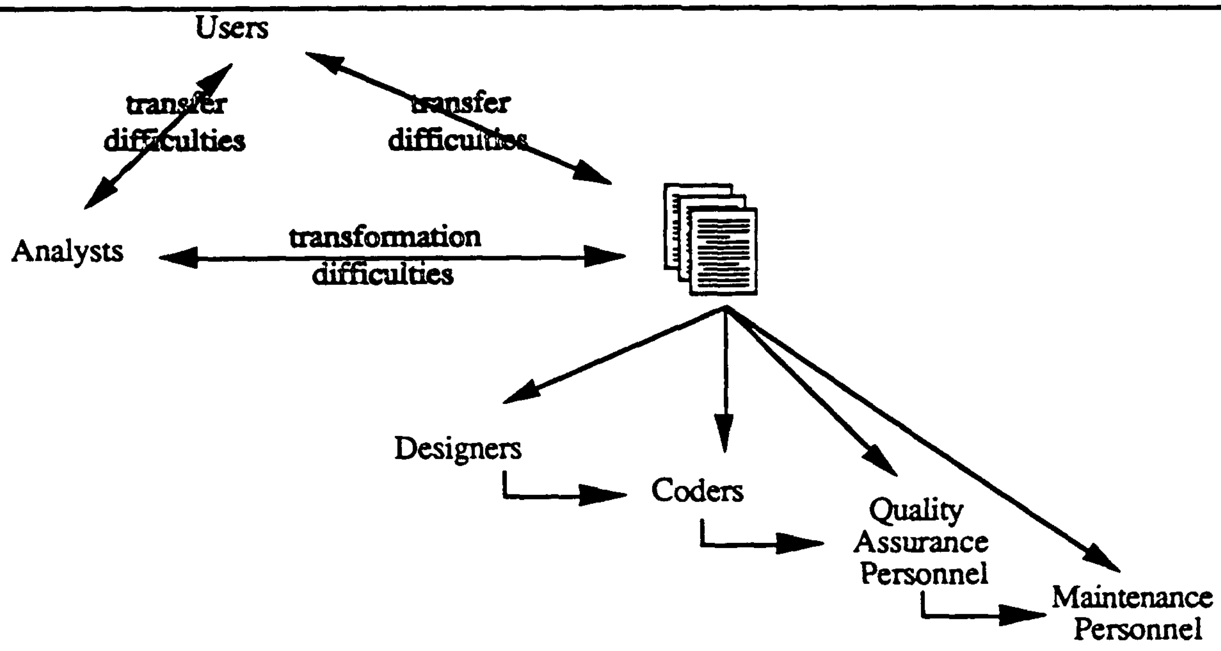


Figure 1: Requirements Information Transfer and Transformation Difficulties

Problems associated with the transfer and transformation of requirements information may be classified into four broad categories. These are briefly discussed in the next section, followed by sections describing our approach to resolution of these difficulties including: the workstation architecture; the system tool kit; and features offering specific support for group interaction. Knowledge-based components of the tool kit include: tools for classifying requirements; analyzing imprecision, conflict, overspecification, and management of volatility; and development and assignment of validation metrics, test tools, and test plans. We close the paper with a brief examination our experience with the workstation as applied to a number of software projects and our plans for future research.

REQUIREMENTS FORMULATION DIFFICULTIES

One way to classify software requirements formulation difficulties is to utilize the following categories: *acquisition, representation, content,* and *analysis.* This classification scheme is based on the necessary processing steps for requirements information during capture and analysis. Each classification category and the apparent difficulties that may arise within them are described below.

- **Information acquisition difficulties.** In situations where the problem is highly complex, the user generally is not able to accurately state requirements for the project. Additionally, the user and the analyst are frequently not able to communicate adequately concerning project requirements. Two factors often combine to keep the analyst from obtaining satisfactory requirements information. First, there are situations that occur concerning complex and/or analytical problem domains for which the user is not able to accurately state requirements for the project. Second, situations occur where users are unable to articulate the requirements in a language the analyst can understand. As a consequence, information will be lost, inaccurately represented or otherwise made less useful. The analyst often has a similar inability to relate to the language of the user and few analysts are trained in areas capable of providing assistance such as: management science, systems engineering, and cognitive psychology (see [Agre86], [Bohe87], [Sage77], [Newe72], [Norm86], [Rasm86], [Schn80]).

- **Information representation difficulties.** Webster and others have noted that an implicit model formed by the analyst contributes to understanding of requirements information [Webs88], [Trea85]. A major problem to be resolved is how to obtain robust representations of requirements information for direct examination during the analysis process. This process, transfer, transformation, and recording of requirements information, is currently accomplished as a manual task. Information must be transformed from its actual representation: dynamic, real-world, and multimedia, into text and graphics. Information will be lost, inaccurately represented or otherwise made less useful. This transformation leads to misunderstandings, as well as summarizing, filtering, omission, and processing errors, all combining to reduce the utility of the information. Thus, analysts are faced with a continuing problem: how to obtain robust representations of requirements information to directly analyze during the analysis process.

- **Information content difficulties.** Requirements information, as provided by the user, tends to suffer from a number of flaws including: fuzziness, imprecision, incompleteness, ambiguity, untestability, overspecification, and internal conflict. Current requirements engineering methodologies do not appear to address these issues. The consequence of this is a lack of support for processes to examine requirements information when checking for flaws in the content.

- **Information analysis difficulties.** System requirements are generally understood to become more reliable, accurate, and trustworthy as development proceeds [Boeh87]. Yet, current methodologies discourage analysts from further enhancing written requirements because they are not able to accommodate unplanned changes or assess the impact of requirements volatility. Additionally, the inability to analyze requirements information in other-than-text-and-graphic-formats limits the available bandwidth for information analysis. This effectively places limits on the amount of requirements information available for analysis due to the narrow bandwidth for information in these formats.

These difficulties account for approximately 80% of the errors found in delivered software [Bohe87]. To address these we have developed a workstation concept that supports: an object management system that facilitates integration of diverse types of information to be utilized during the analysis process; a means of developing reusable requirements information; and a set of knowledge-based tools for increasing the overall robustness of requirements information. We have also provided the means for supporting cooperative group work aspects of requirements engineering. The systems architecture for the workstation and the system tool kit is described in the next section.

WORKSTATION ARCHITECTURE AND SYSTEM TOOL KIT

The workstation is based on an Apple© Macintosh™ fx with 8 megabytes of RAM, 160 megabytes of hard disk storage, and assorted peripherals as depicted in Figure 2. We have developed software supporting basic requirements engineering functions using SuperCard and the SuperTalk language, an extension of Apple's HyperTalk. Both HyperCard and SuperCard are object-oriented programming (OOP) environments. Additional tools were developed using SmallTalk. For more information on the workstation and software see [Aike89] and [Aike90a]. The discussion of the system tool kit focuses briefly on the object management software and then specifically on the integrated set of knowledge-based analysis tools. These tools appear capable of reducing or resolving certain categories of information flaws in software requirements.

Object Management Software

The workstation supports tasks associated with formulating software requirements utilizing a rapid prototyping approach to software development. It supports the integration of traditional text and graphics-based information in combination with full motion video, animation, and sound-based information. The workstation encourages direct access to requirements information with a single access method during analysis. The software provides support for four basic requirements engineering functions: 1) *capturing* information about the task to be performed, systems users, and relevant organizational/situational characteristics; 2) *organizing* information into an easily accessible form for use in subsequent phases; 3) *synthesizing* problem solutions; and 4) *presenting* solutions to the user to obtain corrective feedback. The object management software facilitates the formulation, assembly, and manipulation of multimedia objects depicting requirements information into system specifications through successive refinement [Aike89a]. The system specifications may then be analyzed by use of the knowledge-based analysis tools described in the next section.

Knowledge-based Analysis Tools

We have characterized requirements engineering as a set of tasks involving groups of users and analysts who attempt to provide a concise statement of the needs of the system to be developed and fielded. Requirements information provided by users is often characterized by statements that are ambiguous, inconsistent, conflicting or possess other similar flaws [Sage90]. Software containing these flaws is at risk from the onset and seldom if ever results in the desired outcomes for the software product. Performing manual examination of thousands of individual requirements for these flaws is nearly an impossible task and is often beyond human ability to perform. Analysts lack adequate knowledge-based support for the process of examining the requirements information as a group.

In developing the workstation tool kit we began with the process of bringing order and understanding to the use of quality factor

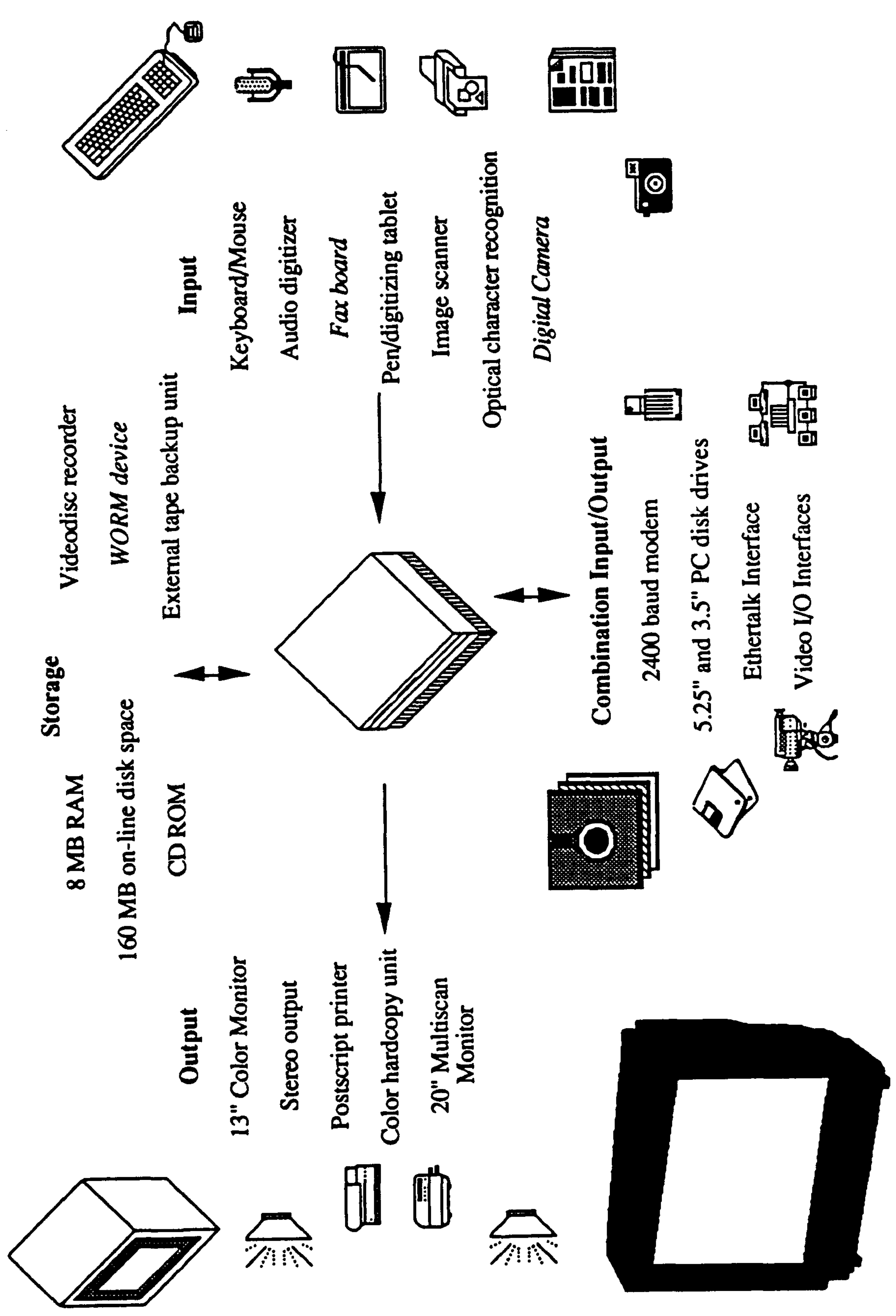


Figure 2 Workstation Configuration

goals in software requirements. We first developed a knowledge-based assistant with the capability of discovering conflict, ambiguity, imprecision, and overspecification in software requirements. Subsequent efforts have resulted in a prototype system for validating software requirements by assignment of metrics, test tools, and finally test plans. We also developed an effort estimation tool to assist with project management in object development contexts [Sams88, Myer88, Palm88a, Phle89]. These tools are described in subsequent paragraphs.

The initial tool was concerned with reducing the difficulties and issues involved with the selection of quality factors by users. Performance goal indicators are often selected by the user without knowledge of the potential for conflict, imprecision, and overspecification. This lead us to the need to classify user requirements, and we developed added features to classify user requirements, search for incomplete or inconsistent requirements statements, and provide a means for handling uncertainty [Palm87]. The outcome of these efforts was a tool capable of taking English language requirements statements, parsing them, classifying them according to functional and non-functional categories, indicating the specific error in the requirements statement, and providing a interactive human-computer environment to manage problem areas and resolve deficiencies. The long-term goal of this research is to provide an environment in support of complex interactive human functioning related to requirements engineering processes [Palm88b].

Experience indicates we are able to classify approximately 60% of the requirements statements presented to the system [Myer88, Sams88]. We are able to correctly identify functional and non-functional requirements statements that are complete, consistent, unambiguous, and precise. The remaining 40% are in such a state as to be non-classifiable and require significant rework before they can be properly classified and be suitable for use in the design process. These statements are classified by the tool according to the type of difficulty present. For example, if imprecise words such as *high*, *medium* , and *low* have been used, or certain statements conflict, the requirement is flagged and the user is asked to resolve them through interaction with the analyst. A statement such as, "the system shall have high reliability," is unacceptable as it fails to communicate to the designer just how high is high and compared to what. The simultaneous selection of *responsiveness* and *interoperability* as performance factors for a system level requirement results in conflict and the user is asked to resolve the issue. Otherwise, the issue is carried through the development process as a problem until it is either resolved or ignored by the development team.

Next we extended this capability to provide for the assignment of metrics to each requirements statement, developed a strategy for assignment of a test tool for the selected metrics, and finally developed a test plan for requirements validation. This tool has been applied to several sets of specifications including the U.S. Army Howitzer Improvement Program (HIP), the Department of the Army Movements Management System-Redesign (DAMMS-R) and a commercial bakery operation. The results from the use of the knowledge-based tools confirm that we have been able to analyze and classify these requirements statements, determine problems that were present, provide the means for interactive resolution of these problems, provide a traceability matrix for these requirements over the entire development process, assign metrics, test tools and develop test plans for validation, and provide documentation of unresolved problems and issues that remain. A conceptual diagram of the interaction of these tools within the workstation architecture and the ways in which they interact to alleviate problems in requirements activities and support alternative design approaches is shown in Figure 3.

FEATURES OFFERING SPECIFIC SUPPORT FOR GROUP INTERACTION

The process of generating requirements for most complex and/or analytical problem domains is almost inevitably a group effort. These efforts can be characterized as follows: several persons get together to discuss the needs of a particular system; they determine the necessary functionality and prepare a document describing system level requirements for the process to be supported. Normal modes of exchange and documentation consist of verbal communication, text reviews, and graphic inputs. The analysis process is usually a cooperative effort, and may be distributed temporally and/or spatially [Krae88].

The workstation concept provides support at the interface between user and analyst groups and for groups of analysts either working together or functioning on a distributed network. Our intent has been to extend both the scope and range of support available to groups of analysts by: 1) supporting functions associated with the process of capturing, organizing, synthesizing, and presenting requirements information as performed individually and in groups and 2) supporting the ability to manage, analyze, integrate, and share information and results from specific requirements engineering procedures. Engelbart has provided the basic objectives of our support for collaborative processes with what he termed augmenting human intellect: 1) gaining situation comprehension more quickly; gaining better comprehension or gaining a useful degree of comprehension can be gained where previously the situation was too complex and 2) producing solutions more rapidly; producing better solutions; producing solutions where previously it was difficult to find any solution at all [Engl62]. In addition, researchers continually stress the value of the application of multiple methods of requirements methodologies within the requirements engineering phase of the system development life cycle [Andr86]. The architecture that we have developed supports different methodologies of choice by the user.

A unique aspect of this form of requirements engineering support is the incorporation of hypermedia-based technologies into the workstation. Users and analysts provide information to the system from a wide variety of sources. Specifically developed software permits the analyst to capture, manipulate, then format, requirements information that exists in a variety of media. They are able to analyze and represent requirements information as it becomes available from non-traditional formats including video, audio, animation, simulation, as well as various combinations of traditional representations including text and graphics. The goal is to provide the user and the analyst with the ability to represent the requirements in the format most appropriate for depicting the system to the user. Another important aspect of this approach is the need to move beyond the narrative form in the extraction of requirements information. Other approaches include video tape of operational situations, audio and video records of interviews for detailed review, and animation to simulate desired outcomes. The capability to do this represents an important departure from currently available requirements techniques.

Hypermedia is a relatively new technology that provides users of computer systems the ability to integrate video, audio, graphics, and text media forms and means to store and retrieve this information. Hypermedia linking capabilities provide the ability to demonstrate connections between prototype design features and at least one requirement. In addition, it provides analysts the ability to manipulate and interact with requirements information in forms close to naturally occurring formats. Specific support for cooperative work includes:

- **Increased information modularity**. Modularization of information simplifies a number of logistical problems associated with object management and facilitates greater utility of information by making it available in related chunks. (A chunk is defined to be all the information in a single record in the workstation. Thus, a chunk could be a simple graphic or an entire video tape.)

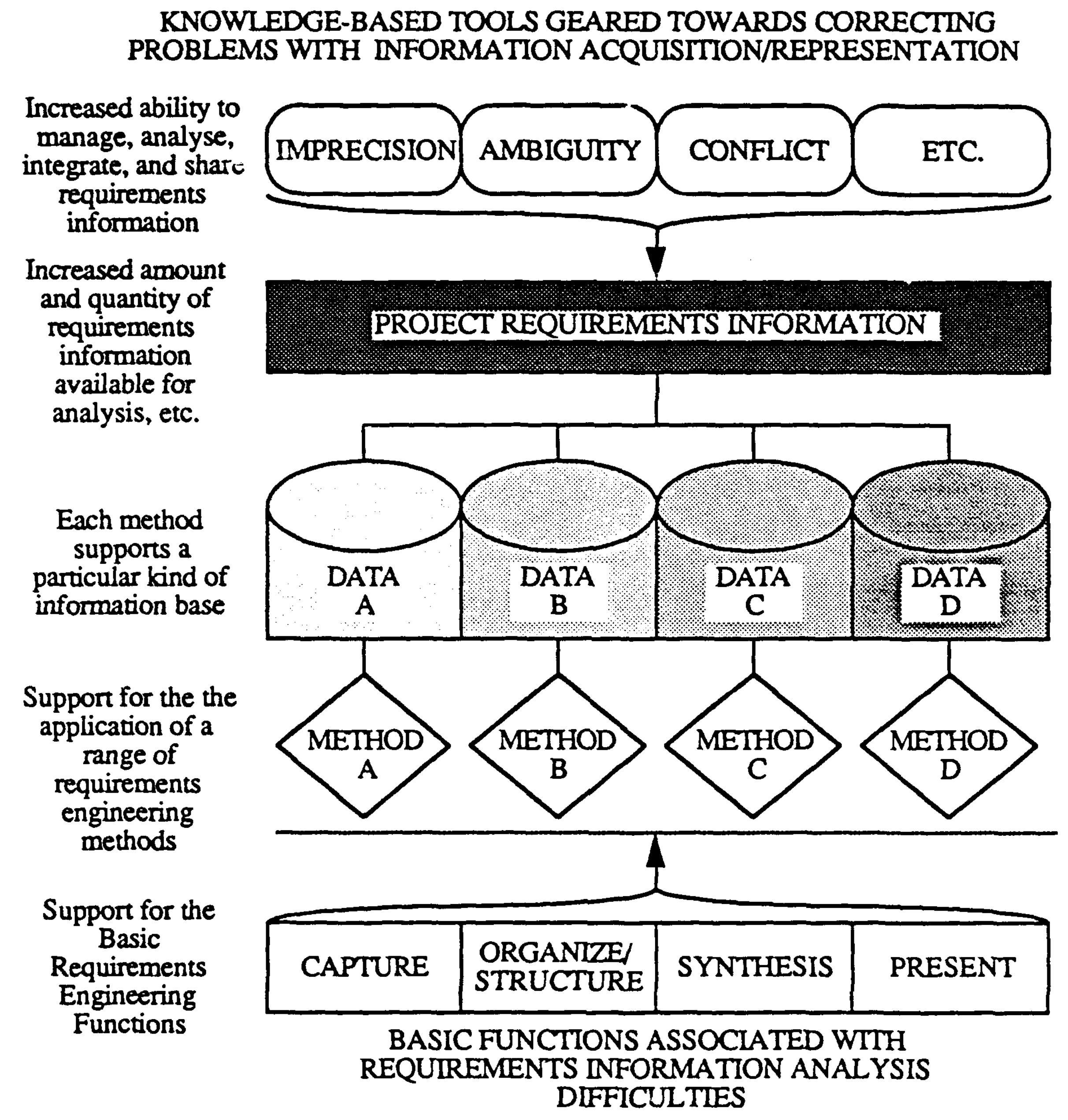

KNOWLEDGE-BASED TOOLS GEARED TOWARDS CORRECTING PROBLEMS WITH INFORMATION ACQUISITION/REPRESENTATION
Increased ability to manage, analyse, integrate, and share requirements information
IMPRECISION
AMBIGUITY
CONFLICT
ETC.
Increased amount and quantity of requirements information available for analysis, etc.
PROJECT REQUIREMENTS INFORMATION
Each method supports a particular kind of information base
DATA A
DATA B
DATA C
DATA D
Support for the the application of a range of requirements engineering methods
METHOD A
METHOD B
METHOD C
METHOD D
Support for the Basic Requirements Engineering Functions
CAPTURE
ORGANIZE/ STRUCTURE
SYNTHESIS
PRESENT
BASIC FUNCTIONS ASSOCIATED WITH REQUIREMENTS INFORMATION ANALYSIS DIFFICULTIES

Figure 3. Workstation Tool Kit Architecture

- **Simplified and integrated access to objects of all types.** The use of object management for the system software design provides easy access to any object (s) stored in the system. This permits analysts to spend proportionally more effort on the analysis processes and less time on the mechanics of the technology.

- **Increased information accessibility.** The ability to store and retrieve multimedia information formats increases the accessibility of this type of information and permits repeated examination of what would otherwise be single access to such information.

- **Enhanced presentation capabilities** The utilization of multimedia capability permits the use of wider bandwidth communication technologies which enhances the presentation capability of the system and makes information available to user and analyst in more readily understandable forms. Video and audio are as easily accessed as text and graphics.

- **Access to external information sources.** The ability to tap external sources of information simplifies cooperation and information exchange operations through unified data structure and software interfaces.

CURRENT RESULTS AND FUTURE RESEARCH

The hypermedia workstation that we have developed in support of an environment for requirements engineering activities has been applied to a number of sets of requirements documents. We have achieved promising results using the workstation that could have been used to improve these software requirements. The results show that we have been able to classify requirements statements; analyze them for issues and problems; diagnose these problems for resolution by the user; assign metrics, test tools, and test plans; and provide supporting documentation for analysis and review of problems and issues with requirements statements.

Other specific objectives satisfied by this approach include:

- reducing or removing barriers to capturing and analyzing naturally occurring multimedia real-world requirements information;

- permitting the application of the "most appropriate" requirements methodology including aspects of computer-supported cooperative work and groupware;

- encouraging the integration and interchange of information contained in the collections of information associated with the separate methodologies;

•supporting an integrated set of analysis tools capable of reducing or resolving problems associated with imprecise, ambiguous, conflicting, or otherwise flawed requirements.

The validity of this approach has been demonstrated with the development of a CSCW group decision support system (GDSS) designed to assist in reaching consensus in complex problems involving regional mobility [Aike90c]. In this work we elicit multimedia information from users, organize and structure the information, and provide for presentation in mixed media formats. This has been particularly useful in aiding groups of individuals become involved in consensus building and conflict resolution relative to problems associated with regional transportation mobility issues.

Research is continuing in the evolution of the workstation concept in several areas. The present need is to demonstrate the feasibility of the workstation concept during the process of eliciting system level requirements. It is our intent to provide the basis for the development of integrated techniques supporting the elicitation, analysis, validation, and maintenance of software requirements. Our initial results are promising however, we have yet to establish definitive outcomes [Aike90b]. We are presently conducting research on human-computer interfaces to determine conditions for optimum information transfer without information overload. Finally, a GDSS experiment is being applied to test the validity of the workstation concept for support of consensus building and decision making.

ACKNOWLEDGEMENTS

This research was supported under contracts from the Virginia Center for Innovative Technology, the U.S. Department of Transportation, Urban Mass Transportation Administration, and by George Mason University.

REFERENCES

Agre86 Agresti, W. W. (Ed.), *New Paradigms for Software Development*, IEEE Computer Society Press, 1986.

Aike89 Aiken P, *A Hypermedia Workstation for Requirements Engineering* (Ph.D. dissertation), George Mason University, 1989.

Aike90a Aiken P, "Hypermedia-based Requirements Engineering," *Software Engineering: Tools Techniques, Practice*, forthcoming.

Aike90b Aiken P, Palmer, J.D., "ReDe: Hypermedia-based Software Supporting Requirements Engineering" Technical Report.- Center for Software Systems Engineering, George Mason University, 1990.

Aike90c Aiken, P., Armour, F., Brouse, P., Fields, A., Hassanpour, M., Liang, J., Palmer, J.D., "Use of Multimedia to Augment Simulation," *Proceedings of the Winter Simulation Conference*, New Orleans, LA., 1990

Andr86 Andriole, S.J., *Handbook for the Design, Development, Evaluation, and Application of Interactive Military Decision Support Systems*, Marshall VA: International Information Systems, 1986.

Bohe87 Bohem, B.W., "Improving Software Productivity," *IEEE Computer*, 20(9):43-57, September 1987.

Enge62 Engelbart, D., *Augmenting Human Intellect: A Conceptual Framework* Summary Report, Stanford Research Institute, on Contract AF 49(638)-1024, October 1962, 134 pp.

Hoga88 Hogan, T. and Swaine, M., "The Great HyperCard Debate," *Bay Area Computer Currents* October 18-October 31, 1988, 6(11):38-43.

Krae88 Kraemer, K.L., King, J.L., "Computer-Based Systems for Cooperative Work and Group Decision making" *ACM Computing Surveys* June 1988, 20(2)115-136.

Myer88 Myers, M., *Quality Factor Assessment in Software Requirements*, PhD Dissertation, George Mason University,1988.

Newe72 Newell, A. and Simon, H. A., *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, NJ 1972.

Norm86 Norman, D. A., and Draper, S. W., (Eds.), *User Centered System Design: New Prespectives in Human-Computer Interaction*, Erlbaum, New York, 1986.

Palm87 Palmer, J.D., "Uncertainty in Software Requirements," Large Scale Systems, Vol. 12, 1987, pp. 257-270.

Palm88a Palmer, J.D., Myers, M., "Knowledge-based Systems Application to Reduce Risk in Software Requirements," *Proceedings, Uncertainty and Intelligent Systems, 2nd International Conference on Information Processing and Management of Uncertainty in Knowledge Based Systems*, 1988.

Palm88b Palmer, J.D., Sage, A.P., "Cognitive Models and User Interfaces for an Advanced Software Systems Engineering Development," Revue Internationale de Systemique, Vol. 2, No. 2, 1988, pp. 195-214.

Pfle89 Pfleeger, S. L., *Cost and Productivity for Objective-Oriented Development*, (PhD. Dissertation), George Mason University, 1989.

Rasm86 Rasmussen, J., *Information Processing and Human-machine Interaction: An Approach to Cognitive Engineering*, North Holland/Elsevier, New York, 1986.

Sage77 Sage, A.P., *Methodology for Large Scale Systems*, McGraw-Hill, New York, 1977.

Sage90 Sage, A.P., Palmer, J.D., *Software Systems Engineering*, Wiley and Sons, New York, 1990.

Sams88 Samson, D., *A Knowledge-based Assistant for Software Requirements Analysis*, PhD Dissertation, George Mason University, 1988,.

Schn80 Schneiderman, B., *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison-Wesley, Reading MA, 1980.

Subc89 Subcommittee on Investigations and Oversight, "Bugs In The Program: Problems in Federal Government Computer Software Development and Regulation" U.S. Government Printing Office, Washington, D.C., September 1989.

Trea85 Treacy, M.E., "Supporting Senior Executives' Models for Planning and Control" in The Rise of Managerial Computing (Rockart and Bullen, editors) Homewood, Illinois: Dow Jones-Irwin, 1986, pp. 172-189.

Webs88 Webster, D.E., "Mapping the Design Information Representation Terrain," *IEEE Computer* December 1988, 21(12):8-23.

BIOSKETCHES

James D. Palmer - BDM International Professor of Information Technology - B.S. and M.S., University of California, Berkeley, 1955 and 1977; Ph.D., University of Oklahoma 1963. Doctor of Public Service (honoris causa), Regis College 1978. He was Director of the School of Electrical Engineering and the Systems Research Center, University of Oklahoma (1957-1966); Dean of Science and Engineering and Professor of Electrical Engineering, Union College (1966-1971); President of Metropolitan State College, Denver, Colorado (1971-78); Administrator, RSPA, U.S. DOT (1978-1979); VP and GM, R&D Division, MTI (1979-1982); and Exec. VP, JJ Henry Company, Inc. (1982-1985). He joined the faculty of the School of Information Technology as BDM International Professor of Information Technology in 1985. He is a member of the Systems, Man, and Cybernetics Society (President, 1976-78), the Computer Society, the American Society for Engineering Education, and has served on the Winter Simulation Board since 1986. He is a registered Professional Engineer in Oklahoma, New York, Colorado, and Wisconsin. Research interests include systems and software requirements engineering and decision support systems. He has been involved in research in adaptive learning algorithms; modeling and simulation of large scale social systems; systems simulation, modeling, and management; and CIM applications. He is the author of three books and more than 75 papers and directs the research of several Ph.D. students.

Peter H. Aiken - received the B.S. (information systems and business administration & management) and the M.S. (business information systems) degrees from Virginia Commonwealth University, the latter in 1984. The Ph.D. degree in information technology was awarded by George Mason University in 1989. Subsequently, he joined the GMU faculty as a Visiting Assistant Professor of Information Systems and Systems Engineering. Major project experience and publications have been in the areas of systems integration and engineering, software engineering, strategic planning, decision support systems, and project management. Current interests include research in the application of hypermedia-based tools and techniques to the process of software requirements engineering and the applications involving multimedia-based support for group decision making.

Requirements Analysis using ARIES: Themes and Examples[1]

Dr. W. Lewis Johnson
USC / Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292-6695
(213) 822-1511; johnson@isi.edu

Mr. David R. Harris
Lockheed Sanders
95 Canal St., Nashua, NH 03061
(603) 885-9182; decvax!savax!harris@ucbvax.berkeley.edu

Abstract

This paper gives an overview of ARIES, the requirements/specification facet for KBSA. ARIES assists requirements analysts, working either alone or in groups, in developing operational specifications of systems. Initial descriptions that analysts give of a system serve as partial specifications, which are gradually transformed and elaborated to produce a complete specification. This approach ensures traceability of requirements, and permits the use of specification validation techniques such as simulation during the analysis process. The following capabilities are provided in ARIES to facilitate this process. A framework for supporting reuse of requirements knowledge has been developed. A sizable knowledgebase has been developed in this framework. Multiple notations and presentations (employing text and/or graphics) of systems are supported, rather than forcing analysts to work entirely in a single formal specification language. A library of transformation operators is used to extend and revise the system description, and to reconcile conflicting views of the system. Analysis capabilities are provided to detect incompleteness and inconsistencies, and to generate behavior simulations. This paper will describe these capabilities, and present examples of their use.

1. Introduction

The Knowledge-Based Software Assistant, or KBSA, as proposed in the 1983 report [7], was conceived as an integrated knowledge-based system to support all aspects of the software life cycle. A number of systems have since been developed as part of the KBSA program, each providing assistance for individual software activities. These include the Knowledge-Based Requirements Assistant for requirements acquisition [10], developed by Lockheed Sanders, and Knowledge-Based Specification Assistant [12, 13], developed at ISI.

The Requirements Assistant provides facilities for acquisition of informal requirements, entered as structured text and diagrams. By recognizing words in a lexicon of domain concepts and by providing hypertext-like support, it assists in the formalization of informal text. It allows users to incorporate reusable descriptions and to manage complexity through the description of system features from different points of view (e.g., data flow, state transition, and functional decomposition). An internal system representation is managed through underlying inheri-

1. This research has been supported by Rome Air Development Center contract No. F30602-89-C-0103. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official opinion or policy of RADC, the U.S. Government, or any other person or agency connected with them. This paper contains no classified or proprietary information. In addition ot the authors, the following are the members of the ARIES project: Kevin Benner, Martin Feather, Jay Myers, Paul Lakowski, and Jay Runkel. Charles Rich and William Swartout contributed suggestions to this work.

tance, default reasoning, and truth maintenance mechanisms. The different points of view are handled as projections of this information much as is done by certain CASE tools to a limited extent (e.g., [9]). In addition, the Requirements Assistant generates DOD-STD-2167A-style requirements documents from its internal system descriptions.

The principal contribution of the Knowledge-Based Specification Assistant was the development of evolution transformations for specification modification [14, 15]. Evolution transformations are operators which modify specifications in well-defined, specific ways. They are organized into a library, indexed according to kinds of semantic changes they perform, e.g., introducing new types or changing data flow paths. Supporting and controlling evolution is a serious problem in conventional requirements analysis; evolution transformations help alleviate this problem. The Specification Assistant provided validation tools in the form of a paraphraser which translates specifications into English [18, 23], a symbolic evaluator for simulating the specification and proving theorems about it [5], and static analysis tools which automatically maintain and update analysis information as the specification is transformed [16].

We discovered that the RA and the SA were addressing many of the same issues, including domain modeling, decomposition and structuring of systems, reasoning about specifications, and support for natural language. We therefore undertook an integration of the two approaches, via a new system called ARIES.[2] In ARIES, the complexity management of the Requirements Assistant and the formal specification constructs of the Specification Assistant are integrated into a single wide-spectrum representation. Systems represented in this manner may be viewed using any of the available presentation media, including graphical diagrams and natural language. The key capabilities of the Requirements Assistant and the Specification Assistant have been combined, and new capabilities are being added, as will be described below.

This paper gives an overview of how ARIES is used, and what capabilities it provides. The emphasis will be on the new capabilities that were not present in the earlier Requirements Assistant and Specification Assistant: descriptions of these systems can be found elsewhere. The vehicle of the overview will be requirements analysis and system design of a particular example system, a traffic light. Section 2 describes the application domains that we have examined, and gives an overview of the development scenario that will serve as a basis for discussion. Section 3 illustrates capabilities for supporting, sharing, and reuse of requirements and domain knowledge. Section 4 discusses the multiple presentations in ARIES and how they are integrated. Section 5 discusses analysis and simulation capabilities.

2. Example Problems

Although the basic mechanisms of ARIES are domain independent, we have been developing them primarily in the context of a specific example domain, air traffic control. Air traffic control is an interesting area for us because of its complexity. A computer system in such a domain must interact with multiple agents, including controllers, pilots, radars, and other computer systems. Furthermore, advances in computer technology provide an analyst with numerous options for embedding a computer into an ATC system. We have been modeling requirements for a particular system in this domain, namely the control system used for air traffic control in the airspace around Templehof Airport in Berlin. We have also studied the requirements for U.S. domestic en route air traffic control systems, i.e., those systems responsible for control of air traffic cruising at the high altitudes reserved for jet aircraft. These requirements are drawn from manuals on pilot and controller procedures (e.g., [2, 3]), and

2. ARIES stands for Acquisition of Requirements and Incremental Evolution into Specifications.

from the experiences of the current FAA Advanced Automation Program [11], whose goal is to develop the next generation of air traffic control systems.

Unfortunately, the complexity of the ATC domain makes it difficult to describe the analysis steps involved in a short space. For a discussion of specification development in the ATC domain, please see [15]. Instead, we will concentrate in this paper on the much simpler but closely related problem of road traffic control. The road traffic control problem shares certain characteristics with air traffic control: both problems are concerned with the maintenance of safe, orderly, and expeditious flow of vehicular traffic. Yet it is small enough that the process of solving it can be easily grasped.

The scenario that we will be describing here is derived from a requirements analysis exercise performed by the ARIES project during the spring of 1990. We were especially interested in how multiple analysts, each with a different view of the system, might be able to collaborate in the development of requirements for a system. Therefore we each worked more or less independently on different aspects of of the problem, and then compared work to see how our different views fit together. Not surprisingly, we encountered serious difficulties in integrating our views, because we were working separately with minimal machine support. However, the exercise did make clear what kind of mechanical assistance would be needed to help coordinate specification development. We will present here an analysis of the same problem, but with ARIES as an active participant.; we will discuss how ARIES alleviates some of the difficulties encountered in the manual development study. The following capabilities will be highlighted in the example:

- sharing of system descriptions,
- reuse of domain and design knowledge,
- support for multiple presentations and formalisms,
- sketching system behavior and validating behavior using simulation,
- modeling requirements as constraints, and using constraint propagation techniques to investigate interactions among constraints, and
- employing evolution transformations to perform and coordinate specification changes, and to implement system design decisions.

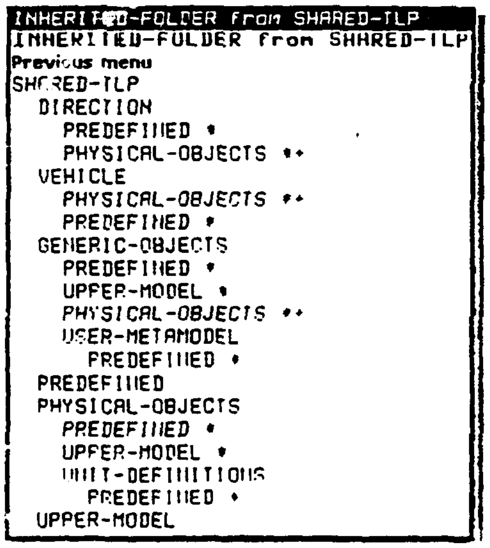
3. Folders, Workspaces and Reuse

It was readily apparent from our investigation of the road traffic control problem that all project participants had a similar intuitive model of the objects, relations, and events in the domain. This model includes concepts such as vehicles, roads, colors, and directions. There were also common notions of system components, including traffic lights and roadbed sensors. In conventional specification development approaches, the specifier must write definitions of these concepts, as well as write requirements in terms of them. Having a library of such concepts available beforehand can reduce the amount of domain analysis and specification effort required.

At the same time, there were key differences in models, depending upon what task an analyst is performing. For example, two distinct models of vehicle motion arose in our road traffic control development. In one, vehicles appear at the entrance to the intersection, traverse the intersection, and then disappear. This corresponds to the information that a traffic light system can have about the environment solely on the basis of what road sensors can provide. In another model, vehicles have a distance from the intersection, a velocity and an acceleration, and approach and depart from the intersection in a continuous process. This latter model was needed to understand the requirements imposed on a traffic light system because of vehicle behavior, e.g., how much time must be allowed between light changes. We needed a way to support such conflicting models, and at the same time understand how requirements stated in terms of one model might be reformulated in terms of another model.

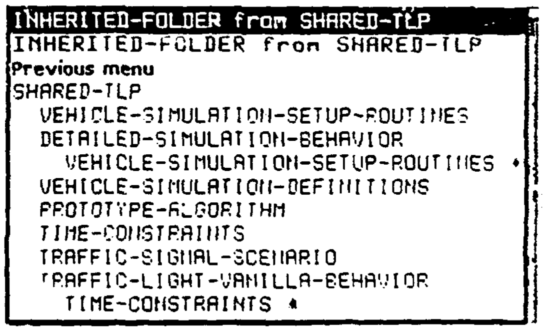
These concerns have motivated two key notions in ARIES: *workspaces* and *folders*. Each analyst interacting with ARIES has one or more private workspaces, which are collections of system descriptions that are to be interpreted in a common context. Whenever an analyst is working on a problem, it is in the context of a particular workspace of definitions appropriate for that problem. In order to populate a workspace, the analyst makes use of one or more folders, which are collections of reusable concept definitions. The ARIES knowledge base currently contains fifty folders comprising over 1200 concepts. Each folder is viewable as an editable presentation of structured text, containing both informal and formal descriptions, viewable as structured hypertext. Reusable formal descriptions include precise definitions of reusable concepts; reusable informal descriptions include excerpts from published documents describing requirements of the domain, e.g., air traffic control procedure manuals.

Figures 1 and 2 give a view of those folders used in the road traffic control problem. The uppermost folder in the hierarchy of folders created for this problem is called `shared-tlp`; this folder contains common domain terminology to be used throughout the project. Figure 1 lists the reusable folders that this folder inherits from. At the highest level are domain-independent descriptions of commonly occurring concepts, such as people, and physical objects, as well as properties that these objects have, such as age, sex, color, and location, and actions that they can perform, such as communicate. General relations that hold between objects, such as equality, greater-than, element-of, etc., are defined at this level. These concepts are captured in the folders `upper-model` and `predefined`. Below these most general folders are folders that give a more detailed taxonomy of objects, including `physical-objects`, which introduces the notions that physical objects have mass, velocity, etc.



```
INHERITED-FOLDER from SHARED-TLP
INHERITED-FOLDER from SHARED-TLP
Previous menu
SHARED-TLP
  DIRECTION
    PREDEFINED •
    PHYSICAL-OBJECTS ••
  VEHICLE
    PHYSICAL-OBJECTS ••
    PREDEFINED •
  GENERIC-OBJECTS
    PREDEFINED •
    UPPER-MODEL •
    PHYSICAL-OBJECTS ••
    USER-METAMODEL
      PREDEFINED •
  PREDEFINED
  PHYSICAL-OBJECTS
    PREDEFINED •
    UPPER-MODEL •
    UNIT-DEFINITIONS
      PREDEFINED •
  UPPER-MODEL
```

Figure 1. Folders inherited by shared-tlp



```
INHERITED-FOLDER from SHARED-TLP
INHERITED-FOLDER from SHARED-TLP
Previous menu
SHARED-TLP
  VEHICLE-SIMULATION-SETUP-ROUTINES
  DETAILED-SIMULATION-BEHAVIOR
    VEHICLE-SIMULATION-SETUP-ROUTINES •
  VEHICLE-SIMULATION-DEFINITIONS
  PROTOTYPE-ALGORITHM
  TIME-CONSTRAINTS
  TRAFFIC-SIGNAL-SCENARIO
  TRAFFIC-LIGHT-VANILLA-BEHAVIOR
    TIME-CONSTRAINTS •
```

Figure 2. Folders that inherit from shared-tlp

Figure 2 shows the folders that inherit from `shared-tlp`. These include two different models of vehicle behavior: a "vanilla" behavior description which simply models vehicles as approaching, entering, and leaving the intersection, and a detailed continuous-time simulation. Below these shared folders are further folders aimed at specific aspects of the problem, being developed by individual project members. One called traffic-signal-scenario models the sequence of state transitions of the traffic light; the one named time-constraints models the timing requirements for the traffic light.

Folders are intended to deal with the problems of sharing and hiding information that arise in systems with large knowledge bases. The conventional approach to information sharing in intelligent systems is through inheritance hierarchies. By placing all concepts in an inheritance hierarchy, concepts that are lower in the hierarchy share properties of concepts that are higher in the hierarchy. Developers of new applications use concepts already in the hierarchy where possible. The problem with this approach is that it is often necessary to model the same concept in different ways, depending upon how the concept is to be used in a particular knowledge-based application. ARIES allows for multiple versions of the same concept to coexist, while capturing their similarities. It also allows individual projects to choose which version of a shared concept will be employed in their project, and further customize concepts as necessary.

An example where a concept is selected and customized for the project as a whole is the following. The ARIES knowledge base contains several alternative models for directions: as compass points (e.g., north, south, east, and west), as the number of degrees clockwise from magnetic north, or as multiples of ten degrees from magnetic north (used to mark the direction of runways). Clearly a model of directions used to mark runways at airports is not well suited for road traffic control problems. In shared-tlp the decision was recorded to model direction in the road traffic control problem as named compass points. This is accomplished as follows. The folder containing models of direction, called direction, contains both a generic concept for direction, also called direction, and each of the various models of direction. The model of direction as compass points is called named-direction. The administrator of the shared-tlp folder inherits the direction folder, and renames named-direction locally as direction. Then whenever project members use the concept direction they will get the compass-point version. ARIES continues to record that the local version of direction is a specialization of the generic direction, so any attributes of that generic concept will apply to the specific concept as well. Thus ARIES supports the process of giving specialized definitions of concepts in specific situations, something which people do constantly when describing domains and systems. It is commonplace in the requirements engineering world to construct domain models from scratch for each system being designed. The ARIES approach makes this unnecessary. Furthermore, by relating project-specific concepts to shared concepts, someone unfamiliar with a project can learn quickly and easily what specific assumptions are being made by the project about general concepts.

In order to further support sharing and reuse, we have developed and extended the notation of specialization hierarchy beyond that used in conventional object-oriented systems, such as CLOS [21] or term subsumption systems such as LOOM [17]. Term subsumption languages provide a precise semantics between types and their specializations. However, no such subsumption relationship holds among relations, or among procedural constructs such as methods. It is possible for two classes of objects to have methods with the same name that do entirely different things, even if one class is a specialization of the other. In ARIES, specialization hierarchies can be defined for relations and events as well as types, and all specializations are defined in terms of subsumption. In the shared-tlp folder, for example, a relation roadway-direction-of is defined as a specialization of direction-of, and is declared to be static, i.e., unchanging over time. Thus,although objects in general can change their direction, roadways cannot. The definition of term subsumption for relations and events involves some technical details that are unfortunately beyond the scope of this paper.

Folders and workspaces make it possible to explore different aspects of a problem simultaneously. This is particularly important when supporting cooperative development by several project members. In the road traffic control problem, there are at least three ways of breaking down the problem. One problem is to determine what the proper sequencing of colors of traffic lights should be. Another is to determine what constraints on the colors displayed by the traffic lights, and on the timing of the traffic lights. A third concern is what the physical configuration of the system should be, e.g., what sensing devices should be used to detect the presence of traffic, and how they should be connected to the controller system. ARIES supports the parallel examination of each concern in a different folder, followed by subsequent integration of concerns.

4. Multiple Presentations

The Requirements Assistant provided supported multiple presentations of the same system description. An engineer could request that the Assistant present the system using any of a number of different diagrams, such as a data flow diagram or a state transition diagram. The KBRA's internal representation of systems captured all of the information necessary to support these diagrams.ARIES extends this idea in a number of significant respects. ARIES's internal representation supports an entire spectrum of descriptions, from informal hypertext to formal specification languages. This makes it possible for ARIES to support the entire specification acquisition process. Furthermore, ARIES supports three different formal languages: Gist, Loom, and Refine [20]. Gist is the specification language used in the Specification Assistant, Loom a commonly used knowledge representation language, and Refine is a high-level programming language developed byReasoning, Inc. Because ARIES is not geared toward a single formal language, it is able to interoperate with other knowledge-based systems, and can easily adapt to future language improvements and standardization efforts. We are examining ways of mixing formal and semiformal descriptions in the same presentations, so that the analysts are not forced to make an abrupt switch from informality to formality. Finally, the ARIES architecture provides support for domain-specific presentations of various sorts.

Multiple presentation capability such as this can be used as follows in the road traffic control problem. Natural language is useful for sketching out in initial statement of requirements, or for acquiring requirements statements from clients. The sequencing of light changes is best expressed in the form of a state transition diagram. Timing constraints, on the other hand, are best expressed in a special constraint language, or in predicate calculus. Designing the flow of information between system components requires a flow diagram or context diagram. Analysis of vehicle behavior in an intersection will require real-time modeling of system behavior, which it turn could require graphical animation showing traffic flow through the intersection over time.

Completing the specification requires the ability to switch back and forth between presentations, and merge aspects of them. Thus, for example, it is useful to generate natural language statements from formalized requirements to check them against the original requirements. Semi-formal notations, such as state transition diagrams, need to be augmented with formal properties. Arcs in state transition diagrams ordinarily are just text strings; in ARIES the transition conditions can be logical predicates on the state of the system. Accordingly, mixed diagrams such as that shown in Figure 3 are desirable. During the specification development process these informal predicates will be replaced with formal logical predicates on objects and relations in the system description. The actual ARIES-generated diagrams will not use Lisp notation, as in this figure, but will use a more conventional notation for predicates.

4.1. A common representation

Our first step was to come up with an internal representation for ARIES that can support these various notations. This task is somewhat more involved than was the case in the KBRA, because the semantics of many of the nota-

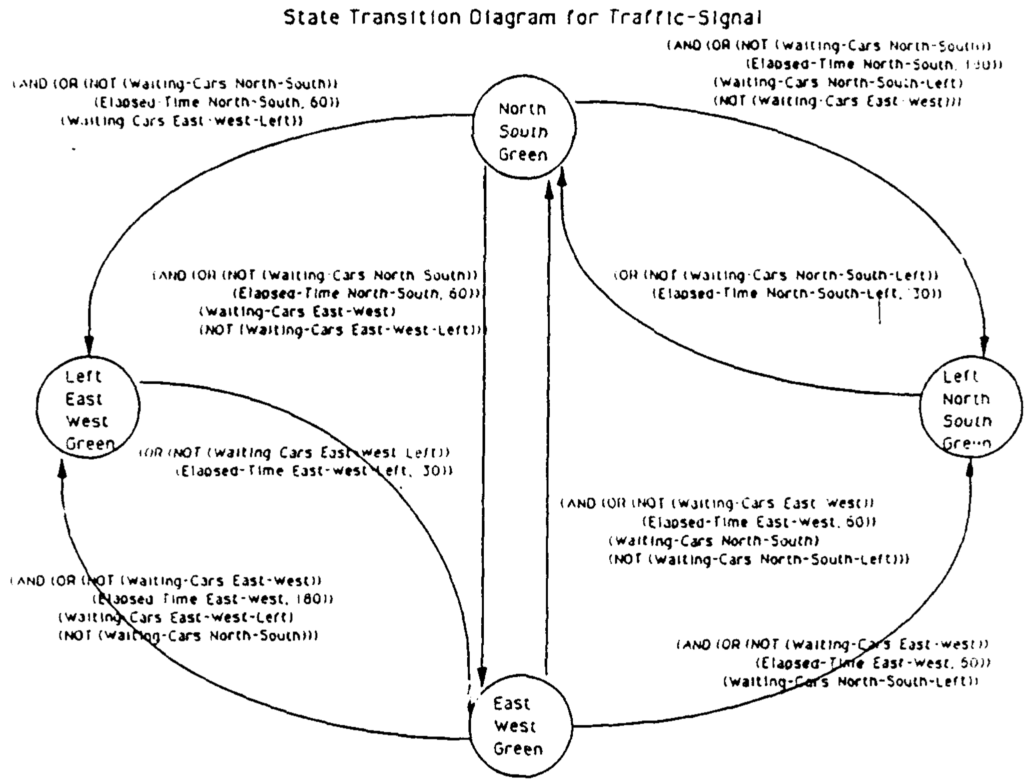

State Transition Diagram for Traffic-Signal
(AND (OR (NOT (Waiting-Cars North-South))
(Elapsed-Time North-South, 130))
(Waiting-Cars North-South-Left)
(NOT (Waiting-Cars East-west)))
(AND (OR (NOT (Waiting-Cars North-South))
(Elapsed-Time North-South, 60))
(Waiting Cars East-west-Left))
North South Green
(AND (OR (NOT (Waiting-Cars North South))
(Elapsed-Time North-South, 60))
(Waiting-Cars East-West)
(NOT (Waiting-Cars East-West-Left))
(OR (NOT (Waiting-Cars North-South-Left))
(Elapsed-Time North-South-Left, 30))
Left East West Green
Left North South Green
(OR (NOT (Waiting Cars East-west Left))
(Elapsed-Time East-west-Left, 30))
(AND (OR (NOT (Waiting-Cars East West))
(Elapsed-Time East-West, 60))
(Waiting-Cars North-South)
(NOT (Waiting-Cars North-South-Left)))
(AND (OR (NOT (Waiting-Cars East-west))
(Elapsed Time East-west, 180))
(Waiting-Cars East-west-Left)
(NOT (Waiting-Cars North-South)))
(AND (OR (NOT (Waiting-Cars East-west))
(Elapsed-Time East-West, 50))
(Waiting-Cars North-South-Left))
East West Green

Figure 3

tions of interest overlap considerably. It was necessary to identify semantic common ground sufficient to handle these notations in a consistent way. This common ground models domains and systems in terms of types, relations, and events. Types and relations correspond closely to the entities and relationships of E-R diagrams, or the concepts and relations of Loom. Events subsume all processes and activities performed by internal system components or external agents, and correspond to the events of ERAE [8], or the demons and procedures of Gist.

Some of the diagrams, such as the taxonomic hierarchy diagrams, map onto the internal representation in an obvious way. State transition diagrams and data flow diagrams pose particular stylistic problems, however. Semantically the mappings are straightforward; for example, we have defined an algorithm which maps the states and transitions of a state transition diagram in a consistent way onto relations and events in our semantic model. It is another matter to look at an arbitrary system description in the internal representation and decide what ought to appear as a state, a transition, a process, etc. Our solution is to rely upon the analyst to provide the cues. If the analyst introduces a concept via a state transition or data flow diagram, ARIES records that the concept is suitable for presentation in that diagram, and will continue to show it in updated versions of the diagram as the system description evolves. In order to diagram concepts that were entered by other means, it will be up to the analyst to describe the class of concepts that he wants depicted. And even then it may not be feasible for ARIES to determine what concepts to show without reliance on theorem proving capability, to show, for example, that a particular event causes a state change.

We have been modeling our semantic representation of system descriptions in terms of the types of components of the descriptions, and the relations between them. Thus we can use the model to model itself. The resulting model is called the ARIES Metamodel. The ARIES Metamodel is realized in ARIES as two folders: one, called `user-metamodel`, contains those concepts that an analyst may need to be aware of, such as relations and events. The other contains concepts that are only needed for internal purposes, and never appear directly in any presentation. Making the metamodel visible to the user helps address the criticism of Abbott, a member of the KBSA Consortium who evaluated the KBRA, that the KBRA has no vocabulary for describing system descriptions [1]. The ARIES Metamodel has been used make the new Gist Paraphraser paraphrase Refine programs, as described by Myers and Williams in this volume [19]. Because the Paraphraser operates on the Metamodel, and the Metamodel is able to represent Refine programs, it was possible to port the Paraphaser into a Refine environment with virtually no modification.

Once the common underlying representation was defined, we developed a new notation for specifications for RG (for Refinable Gist), which employs a combination of Gist and Refine constructs to present system descriptions more clearly than either Gist or Refine can. We are currently exploring the possibility with Andersen Consulting to use RG in the KBSA Concept Demonstration, since they have a need to extend Refine to include some of the high-level specification constructs of Gist and Loom. If this activity is successful, RG and the ARIES Metamodel will have made a significant contribution toward commonality and interoperability within the KBSA community.

4.2. The presentation system

The ARIES presentation system is an architecture for defining interactive presentations linked to the ARIES Metamodel. It is implemented in CLX and CLUE, on top of X windows, and is operational on both the TI Explorer and the Sun. We have currently built state transition diagrams, object type taxonomies, Gist, and concept description presentations in the presentation language. In the upcoming months we will be adding a browser (i.e., a form-based presentation that provides access to all parts of the system description), an information flow diagram, a context diagram, an entity-relationship diagram, and one domain-specific diagram for the road traffic control domain and the air traffic control domain. Each presentation description includes a declarative description of the metamodel relations which are used to establish and link presentation pieces, and the editing and navigation actions (associated either with a presentation piece or the entire presentation). All editing actions result in sending a change description to an "activities coordinator". The activities coordinator, in turn, invokes evolution transformations to perform the changes. This architecture makes use of a new representation of the effects of evolution transformations [15].

The ARIES presentation framework makes it possible to construct powerful presentations combining text and graphics generation capabilities. It is possible to generate a diagram of a system, and at the same time generate an English description of what that the diagram depicts. It is intuitively obvious that diagrams are much more useful with accompanying commentary (such as appears in this paper!); nevertheless, automated tools frequently overlook this point. We are experimenting with such mixed presentations in ARIES. Figure 4 shows such a mixed presentation. The analyst has asked to see a description of the meaning on the part-of relation used in the road traffic control problem. The system looked at the system description and found two objects that take part in the part-of relation, and which could be depicted in a domain-specific presentation for the road traffic control domain. A diagram was composed, and at the same time the Paraphraser was used to generate text describing the situation. Although this diagram does not yet use the X-based primitives, it relies upon the other capabilities of the presentation system. In particular, the presentation system defines an architecture for associating presentation methods for different classes of objects. This association is declarative, so that the system can reason about what kinds of objects are presentable in a presentation when deciding what objects to present.

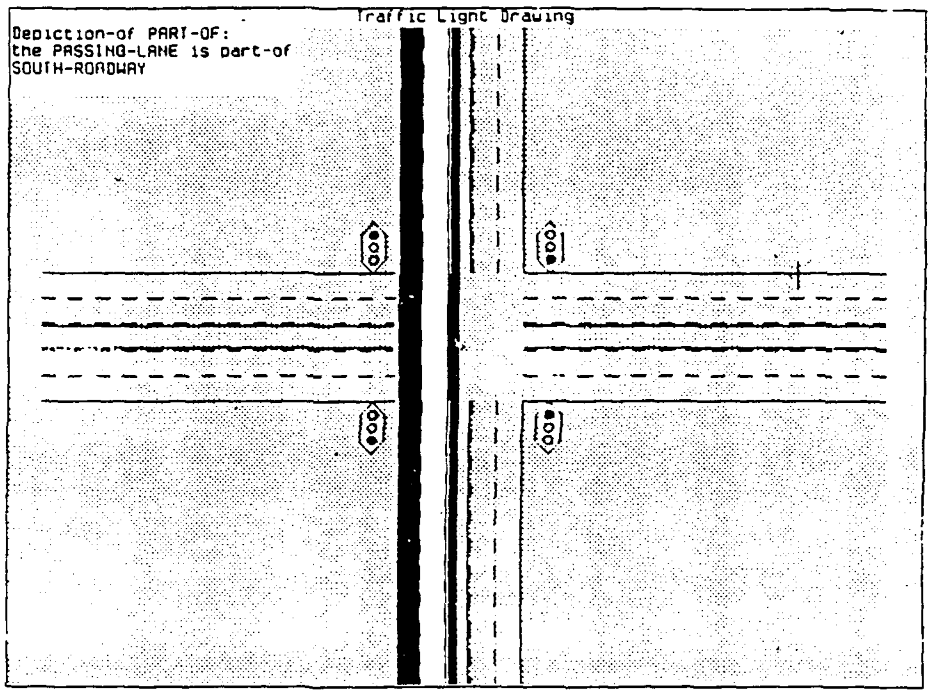


Figure 4. A mixed, domain-specific presentation

5. Simulation and Analysis

Analysis tools are important in order to check for completeness and consistency. A constraint mechanism, derived from Steele's Constraint Language[22], has been incorporated in ARIES for general maintenance of constraints. This mechanism is essential where there are interrelated design properties (e.g., interplay between performance characteristics) and developers can use assistance in identifying when an interaction of requirements may not be achievable. An incremental static analyzer, a version of the static analyzer developed in the Specification Assistant [16], maintains calling and type information for the system description as it is being edited. It also does such things as detect specification freedoms which must be removed temporarily before simulation can be performed.

Simulation tools are useful in order to observe the behavior of a proposed system or its environment, in order to determine appropriate parameters for requirements or to discover unexpected or erroneous behavior. Simulation of vehicle behavior demonstrates, for example, how long it takes for traffic flow to return to normal after a light has changed, thus suggesting what the appropriate light duration should be based on the rate of traffic flow.

Simulation is currently provided by means of a specially modified compiler which translates a subset of ARIES Metamodel into Lisp and AP5 [6]. AP5 is a set of programming extensions to Lisp developed at ISI. Events described in the specification can compile either into ordinary Lisp functions or into tasks to be scheduled by the simulator's task scheduler. Functional requirements in the form of invariants are compiled into rules which notify the analyst if and when they are violated. For more information, see Benner's paper in this proceedings [4].

References

[1] D.A. Abbott. KBSA's Requirements Assistant and aerospace needs. *Proceedings of the 4th Annual KBSA Conference*, Rome, NY, 1989.

[2] Air Traffic Operations Service. *7100.65F: Air Traffic Control*. Federal Aviation Administration, 1989.

[3] ASA, Inc. *Airman's Information Manual*. Aviation Supplies and Academics, 1989.

[4] K. Benner. Using Simulation Techniques to Analyze Specifications. *Proceedings of the 5th Annual Knowledge Based Software Assistant Conference*, Rome, NY, 1990.

[5] D. Cohen. Symbolic execution of the Gist specification language. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 17-20. 1983.

[6] D. Cohen. *AP5 Manual*. USC / ISI, 1989,

[7] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich. Report on a Knowledge-Based Software Assistant.. In *Readings on Artificial Intelligence and Software Engineering*, Morgan Kaufmann, Los Altos, CA, 1986.

[8] J. Hagelstein. Declarative approach to information system requirements. *Journal of Knowledge-Based Systems*, vol. 1 no. 4, Sept. 1988.

[9] D. Harel, H. Lachover, A. Naamad, A. Pneuli, M. Politi, R. Sherman, and A. Shtul-Trauring. Statemate: A working environment for the development of complex reactive systems. In *Proceedings of the 10th Intl. Conf. on Software Engineering*, 1988.

[10] D. Harris. The Knowledge-Based Requirements Assistant. *IEEE Expert*, Winter 1988.

[11] V. Hunt and A. Zellweger. The FAA's Advanced Automation System: Strategies for future air traffic control systems. *IEEE Computer*, vol. 20 no. 2, Feb. 1987.

[12] W.L. Johnson. Overview of the Knowledge-Based Specification Assistant. Proceedings of the 2d Annual Knowledge-Based Software Assistant Conference, Rome, NY, 1987.

[13] W.L. Johnson et al. Knowledge-Based Specification Assistant: Final Report. RADC tech report, 1988.

[14] W.L. Johnson and M.S. Feather. Building an evolution transformation library. *Proceedings of the 12th Intl. Conf. on Software Engineering*, Nice, France, *1990*.

[15] W.L. Johnson and M.S. Feather. Using evolution transformations to construct specifications. In M. Lowry and R. McCartney, eds., *Automating Software Design*, AAAI Press, 1990.

[16] W.L. Johnson and K. Yue. An integrated specification development framework. In *Proceedings of the 3d Annual Knowledge-Based Software Assistant Conference*, Rome, NY, 1988. Also available as ISI tech report RS-88-215.

[17] R. Mac Gregor. *Loom Users Manual*. USC / ISI tech report, 1989.

[18] J.J. Myers and W.L. Johnson. Towards specification explanation: Issues and lessons. In *Proceedings of the 3d Knowledge-Based Software Assistant Conference*, Rome, NY, 1988.

[19] J.J. Myers and G. Williams. Exploiting metamodel correspondences to provide paraphrasing capabilities for the Concept Demonstration. *Proceedings of the 5th Annual Knowledge-Based Software Assistant Conference*, Rome, NY, 1990.

[20] *Refine User's Guide*. Reasoning Systems, Palo Alto, CA, 1986.

[21] G.L. Steele Jr. *Common Lisp: The Language*, second edition. Digital Press, 1990.

[22] G.L. Steele Jr. *The definition and implementation of a compouter programming language*, MIT AI TR 595, 1980.

[23] W. Swartout. The Gist English generator. In *Proceedings of the National Conference on Artificial Intelligence*, Morgan Kaufmann, Inc., 1982.

A CHANGE AND CONFIGURATION MANAGEMENT MODEL FOR THE KBSA FRAMEWORK[1]

John Kimball
Aaron Larson
Honeywell Systems and Research Center
3660 Technology Drive
Minneapolis, MN 55418-1006

jkimball@src.honeywell.com
alarson@src.honeywell.com

ABSTRACT

Change and configuration management (CCM) is a set of abstractions, techniques, and tools for managing the evolution of systems of interdependent design objects. The KBSA Framework's CCM model encapsulates our understanding of the approach required to support CCM in a KBSA-like context. Change is not represented by copying individual objects, but rather by viewing a configuration as a state repository capturing the changes which occurred during one design transaction; all accesses to objects' state must occur in the context of some configuration or proto-configuration. A *design transaction* is an atomic long-duration set of operations performed by multiple agents on a proto-configuration; it starts with one consistent configuration and yields another. Configurations are annotated with *compatibility attributes*, indicating which versions of objects may be substituted for each other; these record the results of static analysis, testing, and experience. A *configuration schema* describes how to construct or recognize a consistent configuration built from acceptable components. It may include cross-configuration references (*version cursors*); *dynamic* version cursors specify a search rule for locating an acceptable source configuration. An in-house project has implemented part of the model to solve CCM problems in an avionics design capture system; their initial experience is described.

1. INTRODUCTION

Design objects – hardware designs, software modules, document sections – are typically highly interdependent. For example, a document is often produced as multiple files of text and diagrams, including reused boilerplate. Figure 1 shows a hypothetical document, consisting of four text objects (*.tex) and three graphics files (*.ps). Cm.tex has three components on which it depends, intro.tex has one component, and model.tex has two. (The document has other dependencies not depicted – dependencies on a typesetting program which accepts input of a particular format, etc.) Likewise, a software system typically consists of multiple modules, reuses routines from various libraries, and relies on the services of the operating system, various daemons, etc.

Design objects are also subject to repeated change over their lifetimes – to correct errors, to cope with changing requirements, or to add new features. We call the various states in the evolution of an object the *versions* of that object; each state (except the first) is a descendent of some other state(s), and may be an ancestor of still others.

For any given use of a particular design object (or system of design objects), some versions of

[1]This work was supported by the Rome Air Development Center under U.S. Air Force contract No. F30602-86-C-0074

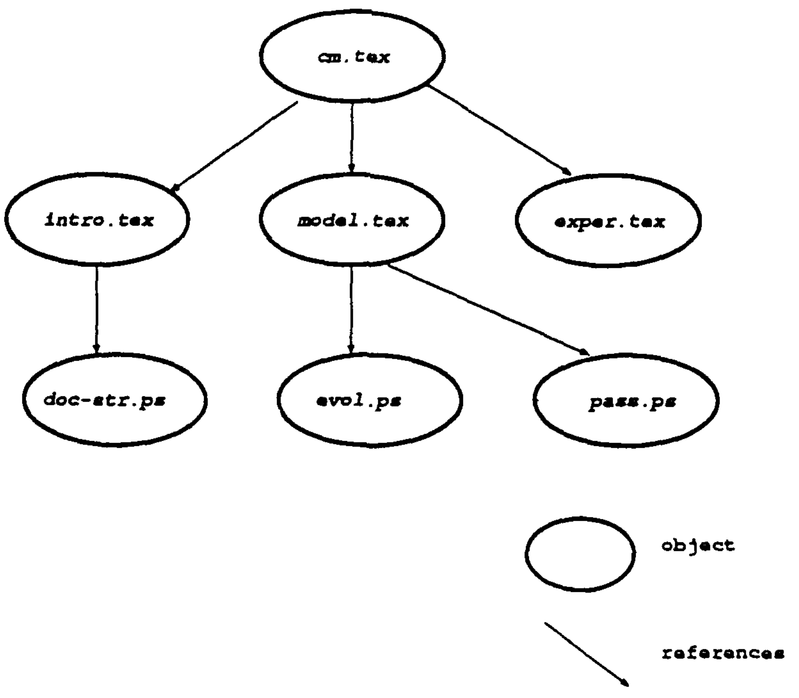


Figure 1: A document and its components.

that object (or system) will satisfy the need, and other versions will not. Text file "exper.tex" needs to be updated with new information; behavioral model "ALU_C9" has a particular bug which the current application exercises; network server "dump-daemon" uses a protocol different from the one required. The phrase "version N is required" – as in "X11 release 4 is required" – is ubiquitous.

These characteristics of design objects – their interdependency and tendency to change – are the motivation to find techniques and tools to manage change effectively. When an object depends on a web of other objects, and many of these objects change over time, then determining and maintaining consistency, compatibility, and equivalence becomes a problem. Whenever such a system must be changed in any way, its massive complexity is a dangerous occasion for confusion and error.

Change and configuration management (CCM) is a set of abstractions, techniques, and tools which assist in managing the evolution of systems of interdependent design objects. Over time, change occurs repeatedly as the systems of objects are modified by multiple agents. The systems of objects have class- and application-specific definitions of consistency.

We have developed a CCM model for the KBSA Framework, which encapsulates our understanding of the approach required to support CCM in a KBSA-like environment. This paper summarizes that model, described fully in [KBSAFW]. It also describes the experiences of an in-house project which has implemented part of the model to solve CCM problems in an avionics design capture system.

2. CONFIGURATIONS AS CONTEXTS FOR STATE

CCM is a global issue; it cannot be dealt with locally, on an object-by-object basis. We must determine how to represent change for CCM, given that

- The objects which are changing are interdependent.
- We must support both forward and backward changes (recoverability).

Many models and systems represent change by copying. When a version of an object is checked-in, that copy of the object becomes read-only. When there is a need to modify the object (culminating in a new version), a writable copy "of the same name" is created (check-out). This strategy works reasonably well in the domain of files and filesystems; but in highly interconnected and interdependent object-bases it quickly becomes unmanageable, due to issues of object identity.

1. Consider the object being copied. When we copy an object, we do not necessarily want to simply duplicate the object references within that object (*shallow copy*); we may need to duplicate the objects referred to by that object, and have our new object refer to those duplicates instead (*deep copy*).

   Suppose cm.tex (Figure 1) was previously checked-in, and we now want to modify it - ie, we need a new version. If we represent change by copying, then we do not want just a copy of cm.tex - we want its components, too. For instance, if we should decide that we must modify cm.tex's component exper.tex, we don't want to modify the same exper.tex referenced by the checked-in version - we would be modifying the checked-in version simultaneously, as a side effect.

   Because the *.ps objects are transitive components of cm.tex, we will want copies of them, too. But suppose that the *.ps objects included a slot which identified the drawing program which produced them. We do not (normally!) want to cause new copies of the drawing program to come into existence when the *.ps objects are copied; we want the new *.ps objects to refer to the same drawing program as the old copies. Thus, we want some slots to obey deep-copy semantics, but some to obey shallow-copy semantics.

2. More problematically, consider not the object being copied, but the objects which refer to it. The slots which previously pointed to the older copy *may* need to be readjusted to point to the writable copy, and the objects where those references occurred *may* themselves need to be copied (since they are being implicitly changed via changing objects which are in their slots). This is the *change propagation* problem.

   In Figure 1, cm.tex references model.tex. Model.tex references evol.ps and pass.ps. Assume that model.tex has been checked-in, and we decide to change it. We must make a new copy of model.tex in which to make the change, so that we maintain history. But model.tex is a feature of cm.tex, and we've changed that feature; it has therefore changed, so perhaps we should make a new copy (a new version) of it, too, pointing at the new model.tex. Likewise, we may need to copy the parents of cm.tex, and their parents, transitively. Further, suppose some other document has chosen to reuse cm.tex (and its components). If we change model.tex in the current document, what, if anything, should happen to the other document?

Because of these problems, our model does not represent change by copying. We contend that if an object is to exist in more than one configuration – ie, have more than one version – then all references to the state of an object must be made in the context of some configuration. If objects may have multiple states, data access by (object, slot) no longer makes sense. Data access must also *in all cases* specify the configuration in which the access is to be performed: (object, slot, configuration). This specification of a configuration can be implicit, but it must occur. Therefore, in our model, we consider a configuration to be primarily a context for determining the state of objects; it is a repository of state. A configuration records the state of the objects which are mentioned in it, not the objects themselves; it maps object $\times$ slot $\rightarrow$ value. It can be considered to be a table of (object_id, slot_id, value) tuples.

We are thus versioning *sets* of objects – configurations – rather than individual objects. If object A exists (is mentioned) in configuration :C:, then the state of A in :C: is called a *version* of A; object A in a different configuration is a different version of A. Versions of objects exist only in configurations; objects under CCM have no state outside of a configuration. This gives us the same semantics as deep copy, but with a space-efficient representation, and a conceptual simplicity.

2.1 CONFIGURATIONS AS DELTAS

A configuration need only record the *changes* in state which were made during one design transaction – a successor configuration is a delta from its predecessor. This leads to a space-efficient representation of a tree of configurations; with appropriate design, the representation can also be time-efficient.

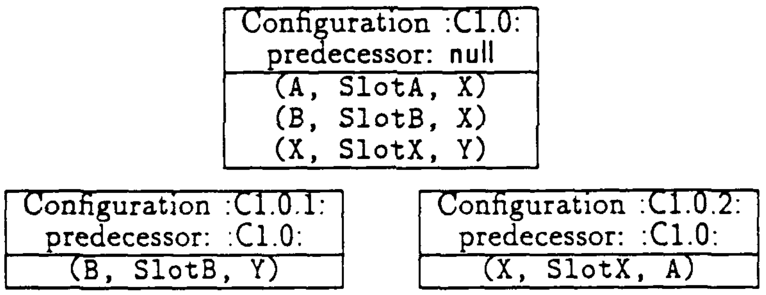


Figure 2: Configurations :C1.0.1: and :C1.0.2: as deltas from :C1.0.1:.

A history DAG is a directed acyclic graph of configurations which share a derivation history; the configurations are related by ancestor/descendent links. In our model, we restrict the history DAG to be a tree – ie, we disallow multiple predecessors (parents); we model multiple predecessors as one predecessor plus other donor configurations "loaded in". Because of this, a configuration need only record *changes* in state – a successor configuration is a delta from its (single) predecessor.

In Figure 2, we see configuration :C1.0: and its child configurations, :C1.0.1: and C:1.0.2:. We assume that :C1.0.1: was produced by

1. Checking-out :C1.0:, and
2. Making one change: (setf (slotB B) Y).

and that :C1.0.2: was produced by

1. Checking-out :C1.0:, and
2. Making one change: (setf (slotX X) A).

Only the *changes* need to be recorded in the child configuration. When it is necessary to read the state of unchanged objects, we traverse the predecessor link (recursively, if necessary). So, for instance, in :C1.0.1:, the value of (SlotB B) is Y, and the value of (SlotA A) is X (retrieved from :C1.0:); in :C1.0.2:, the value of (SlotX X) is A, and the value of (SlotB B) is X. This gives the same semantics as deep copy, but with a space-efficient representation.

This example uses "forward diffs" – the predecessor is treated as the original, and the successor as the revision. For a representation which is not only space-efficient but also time-efficient on average, "backward diffs" can be used as in [RCS]: the successor stores the complete state of the system of objects, and the predecessor's table indicates the changes which must be made to the successor to restore the predecessor.[2]

In version-control systems like RCS, the version history of a file is recorded in a single file; from that version-history file, any version of the file under revision control can be extracted. Versioning is thus done on a file-by-file basis, analogous to modeling change by copying. Our configurations would be similar to version history files maintained on *systems* of files, rather than on individual files; they would be, in effect, a combination of "patch" files – which describe deltas on an entire system of files – and RCS ",v" version-history files.

2.2 COMPATIBILITY ATTRIBUTES

Besides objects' state, configurations also include *compatibility attributes*, which annotate the history DAGs with *satisfaction graphs*; this information is used when de-referencing version cursors.

Certain objects are equivalent to one another with respect to particular operations in particular circumstances; a central task of CCM is to know – to track or to determine – which objects are equivalent, with respect to particular operations in particular environments. Version equivalence and compatibility must be considered in the context of a given operation in a given environment – pcl-v7 and pcl-v8 may be source-compatible (compatible with respect to compilation), but not binary-compatible (compatible with respect to linking). A degree of equivalence and compatibility information can be determined formally, from the structure of objects; for instance, two objects with radically different protocols are not likely to be equivalent. But in general, testing and experience must be used to make such determinations. The inability to retain and use such information (including automated use) is a major limitation of existing systems. In our model, *compatibility attributes* can be set on configurations to record the knowledge gleaned from such testing and experience. In our model, compatibility attributes are part of the data making up a configuration.

Compatibility attributes define the set of directed satisfaction graphs. A compatibility attribute can be considered to be at least a labeled directed arc between two configurations.

[2]A production CCM system would also have to address compaction and garbage collection issues. It would be useful to have the capability of occasionally identifying configurations in the history DAG which are no longer worth saving; these could be compacted out of the DAG, by migrating their changes into preceding or succeeding configurations.

The label specifies the operation and environment under which the two are compatible, and the direction specifies the subset relationship between the two configurations' satisfaction sets. The exact format of the compatibility information, the model of incompatibility, and the format of version-selection rules which use such information needs to be further determined.

3. TRANSACTIONS AS CONTEXTS FOR STATE

The initiation of a design transaction creates a *transaction handle* or proto-configuration. The transaction handle is a work context; it is a writable configuration – a repository for the state of a set of objects under CCM. Transaction handles are long-duration, sharable, and atomic; they may be nested to provide a hierarchy of workspaces. When the transaction is successfully committed, the state of the transaction handle becomes read-only: the transaction handle becomes a configuration, guaranteed to be a consistent system of objects, based on class- and application-specific definitions of consistency.

A design transaction is an atomic, long-duration sequence of operations, performed by multiple agents on a shared system of objects, which starts with one configuration ("check-out") and yields another consistent configuration ("check-in"). In the KBSA framework CCM model, the operations making up a design transaction are performed on a transaction handle – a writable proto-configuration. The transaction handle is a first-class object, which can be passed around and shared. The handle is shared by the agents performing the transaction; users who have its handle, and write-permission to the handle, can perform operations within the transaction. When the transaction is committed, the transaction handle is checked for consistency; if it passes the consistency-checks, the transaction handle becomes a configuration – its state table becomes read-only. Committing a transaction causes the proto-configuration's dynamic references to become fixed; all dynamic references within the configuration's state table become static references.

Though the state-tuples in its state table are frozen, certain modifications can be made to a configuration after commit.

- Compatibility attributes can be added, modified, and deleted, to record the results of testing and experience. Similarly, change-request annotations can be added, describing bugs, deficiencies, and wish-lists.

- It may be possible to add new secondary representations, depending on the purpose of the configuration.

  Design objects are frequently multi-representational. The same conceptual object is depicted by several different representations, often at different levels of abstraction. For example, an ALU hardware design may be represented by a layout object and a netlist object; a program may be represented by a spec object, a source code object, and an object-file object. Representations are related to each other by transformations. A *primary representation* is produced with human input. A *derived representation* is generated from another representation by application of a transformation (eg, the compile transformation, which derives relocatable-object from source-code); if the transformation is purely mechanical, the derived representation is also a *secondary representation.*

  If the committed configuration is only for change management, then the addition of new secondary representations can be allowed. If the configuration is intended to be a release for a particular platform, the presence or absence of secondary representations is significant, and the addition of new secondary representations will probably be disallowed.

3.1 SUCCESSOR TRANSACTIONS AND SUBTRANSACTIONS

The DAG of configurations grows as design transactions are applied to existing configurations. A design transactions which starts with an existing configuration and yields a successor configuration can be called a successor transaction. When such a transaction is initiated, its transaction starts with the same state recorded in the predecessor configuration – ie, the transaction handle's state table is empty. During the transaction, operations are performed, causing object modifications, object creations, object deletions, and/or the import of objects from other configurations; as operations are performed, the state table records the modifications performed to the state inherited from the predecessor configuration.

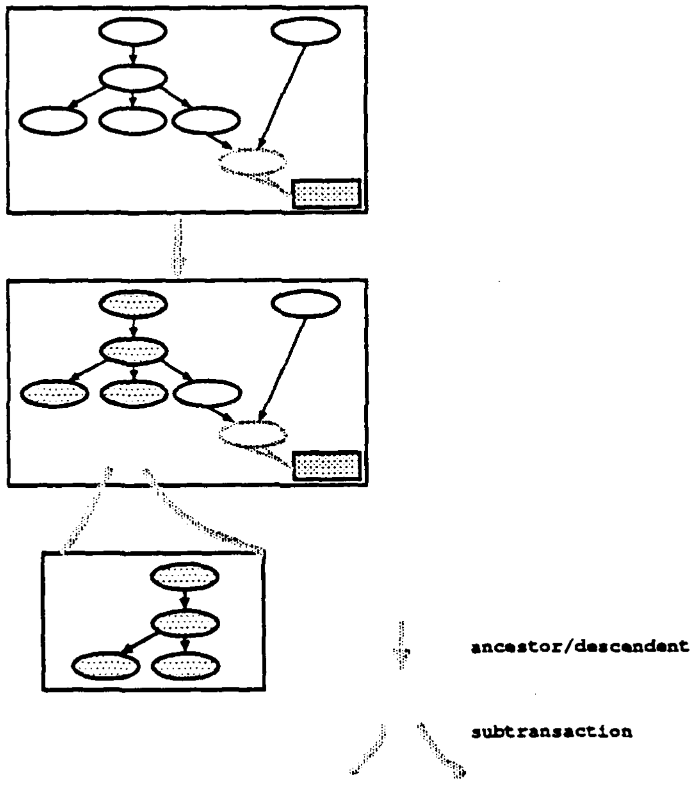


Figure 3: A successor transactions and a subtransaction.

A design transaction can start inside another ongoing transaction, in which case it is a subtransaction. A subtransaction imports objects (actually, their state) from its super-transaction. Often, a subset of the super-transaction's state is imported, since the subtransaction is frequently used to model task decomposition. Committing the subtransaction typically updates the state of the super-transaction with the changes made in the subtransaction.

Figure 3 shows a successor transaction which has one ongoing subtransaction. The subtransaction is operating on a subset of the super-transaction's state.

4. CONFIGURATION SCHEMA

It is useful to be able to specify a configuration abstractly, including component references which are not resolved until configuration-construction time. A *configuration schema* specifies how to build (or recognize) a consistent configuration, and how to correctly propagate change notifications. It is an abstract specification of a configuration, the schema for a consistent system of objects; a configuration is instantiated from a configuration schema. The configuration schema is a set of rules specifying how to build - or how to recognize - the right set of objects, constructed from the right components and utilities.

The configuration schema is part of the configuration - different versions of it may exist in different versions of the configuration. But it is intended that the configuration schema will change more slowly than the configuration itself; it should be more abstract, describing more than one version of the configuration. Further, when the design transaction is committed, all references in its state table become fixed references; however, the configuration schema remains an abstract specification, including the references to objects in other configurations (version cursors).

Typically, a configuration schema will be developed and elaborated over a period of several transactions. It summarizes experience regarding how to properly construct, and properly verify, versions of this configuration. It thus abstracts the activity of a sequence of transactions; it is an abstraction of the "transaction audit trail", replay information. Therefore, the configuration schema specifies not only the structure of the configuration, but also the process to be followed in producing or verifying it.

A configuration schema includes:

Structure description The abstract description of the structure of the configuration identifies the components and utilities required, and how their interfaces are composed together. Objects may be specified by cross-configuration references. For example, in Figure 4, npasswd references ck_pw_lib, which exists in another configuration, ck_pw; mk_db similarly references db from source configuration dbm_lib. Cross-configuration references may be dynamic; resolution of a dynamic reference may depend on the current environment and the desired operation. For example, npasswd references one of two possible be's; be_{fs} (filesystem-based back end) or be_{yp} (daemon-based back end). Selection of the appropriate be can depend on the environment for which npasswd is being built.

Interface description The description of the configuration's interface identifies the objects which the configuration *needs* from other configurations (ie, the version cursors); it also identifies the objects which the configuration *provides* to other configurations (ie, the objects which it is believed should be the external objects, that other configurations may choose to reference).

Verification description The description of how to evaluate the consistency of the configuration may include several things:

- The dependencies between objects, including the dependencies of derived representations (eg, A.exe depends on A.lisp).

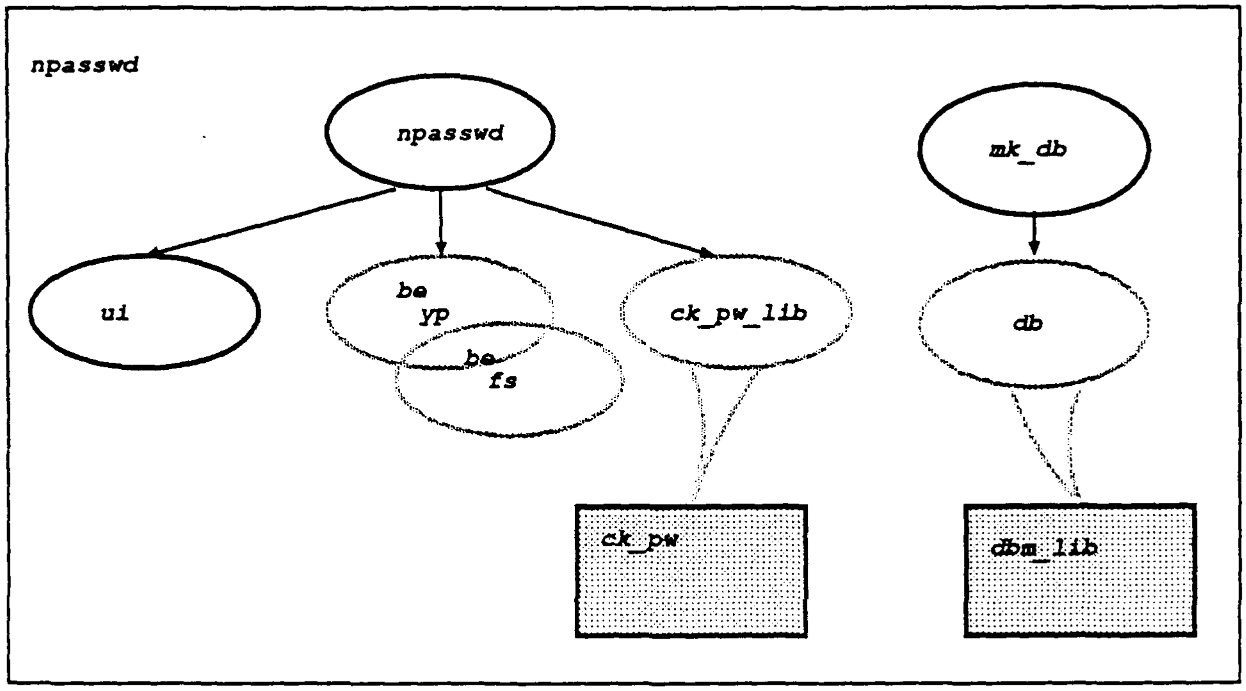


Figure 4: Password-Changer configuration.

- Regression tests, including the use of V&V tools.
- User-specified constraints on the attributes of objects.

Process description The description of how to construct the configuration (ie, to establish consistency) tells how to generate or update dependent objects (including derived representations), and in general how to construct and compose objects which must be constructed or composed. These "actions lines" may be conditionalized on the current environment and the desired operation. For example, in the VMS environment oms.ada would be compiled with a VMS compiler (and a particular system.ada), but if the configuration is being built in the Unix environment, a different compiler and a different system.a would be used. What part of the process description is actually executed at consistency-establishment depends on the areas of inconsistency identified via the verification description.

Change propagation The description of how to propagate change into this configuration from foreign configurations may simply be the interface description's dynamic references. For example, if one of the dynamic version cursors had previously resolved to C:1.9:, but the newly-created C:2.0: would also satisfy it, then by default a change notification message should be sent to this configuration's owner. More generally, it should be possible to specify what action, if any, should be taken in response to a dynamic reference becoming unsatisfied or resatisfied.

A "Makefile" can be considered a weak configuration schema. In its structure description, the only relevant property of the objects are their relative timestamps. No interface description is provided. The action lines of the build rules form the process description; the operations specified there cannot be conditionalized. The only inherent verification description is the build rules, which define consistency based on relative timestamps. Cross-configuration change propagation is not defined, though intra-configuration change propagation is handled by the build rules.

Our understanding of the requirements which must be satisfied by a notation for configuration schemas (and for version cursors) is fairly detailed; we have not chosen a particular notation so far. Further work must be done hypothesizing and testing notations.

4.1 CHANGE PROPAGATION AND CHANGE NOTIFICATION

There are several benefits which arise from our model of configurations and transactions.

Change propagation and deep copy. Because we do not model change by copying individual objects, we avoid the need to do change propagation for check-out operations.

Disambiguating the propagation path. When dependencies among objects form a DAG rather than simply a tree, then there are multiple possible paths by which change can propagate through a system. If all paths are followed, a proliferation of uninteresting or unintended versions occurs. A mechanism for group check-in/check-out allows the dependencies to be disambiguated ([KATZ1]). Our transaction serves as such a delta set (one logical change subsuming multiple physical changes). Thus, other configurations and transactions need only to react to the entire transaction, not to the various intermediate operations within it.

Cross-configuration vs intra-configuration propagation. We can distinguish between cross-configuration change propagation and inter-configuration change propagation, and specify different strategies for each.

Version cursors and configuration schema may be used to control cross-configuration change propagation. For example, if one of the dynamic version cursors had previously resolved to C:1.9:, but the newly-created C:2.0: would also satisfy it, then by default a change notification message should be sent to this configuration's owner. More generally, it should be possible to specify what action, if any, should be taken in response to a dynamic reference becoming unsatisfied or resatisfied. It should be possible to specify the types of changes which should cause a dynamic reference to send a change notification, the types of changes which should be ignored, and the types of changes which should cause action to be taken automatically (eg, a rebuild).

Intra-configuration change can typically be handled by a passive (flag-based) strategy, augmented with user-defined constraints. For many changes, we can postpone resolving the inconsistencies arising from such changes until consistency-establishment or configuration-construction occurs. For some changes, we will want dependent objects to react immediately; user-defined constraints allow such needs to be specified.

4.2 CROSS-CONFIGURATION AND DYNAMIC REFERENCES

A configuration schema may include references to objects which will be fetched from other configurations; the references may be dynamic – ie, the particular source configuration may not be chosen until configuration-*construction time*. A *version cursor* includes (a) the object signature and (b) a rule for selecting an appropriate version of that object (ie, an appropriate source configu.ation). *Dynamic* version cursors provide for flexibility in evolving configurations, and allow references to hypothetical objects which will be constructed in other transactions.

In the configuration schema, when describing a component, we may specify that it should be imported from a foreign configuration (ie, it is not or may not yet be in the current transaction). A reference to a component in a foreign *configuration* – a *cross-configuration* reference – is a *version cursor*. The version cursor may be static or dynamic. Since it must resolve to a particular version of an object, it must specify:

The object's signature. This may be a name, an interface or protocol specification, or something more complex.

A version-selection rule. This may be "version :1.5:" (for a static cross-configuration reference), or a more-or-less complex dynamic reference – eg ":binary compatible with currently loaded windowing system:" The form of version-selection rules depend heavily on the form of compatibility attributes.

There are a variety of characteristics which could be included in a version cursor's version-selection rule:

A static configuration identifier The selection rule identifies a concrete configuration ("C:2.1:") whe the version cursor is a static reference.

Compatibility requirement We may specify that we want a version which is compatible with some other configuration. The other configuration may be a known concrete configuration, or it may itself be a dynamically-chosen configuration. In the latter case, backtracking may be necessary to discover a consistent (or the "best" consistent) system of configurations.

Change request responses We may specify that we want a version which has responded to particular change requests (ie, which has fixed certain deficiencies).

Environment and operation dependencies We may conditionalize the request on the environment or desired operation, particularly for configurations which are intended to be constructed for multiple environments or to satisfy multiple purposes.

Dynamic version cursors make significant use of compatibility attributes in specifying and resolving version-selection rules. We have described compatibility attributes as being at least labeled directed arcs, which specify that one configuration can satisfy a request asking for another configuration, in the context of a particular environment and a particular operation. We are assuming the version-selection rules are first-order logic expressions. The selection rules and the version-search process must be adequately powerful; but if the expressions become too general, then the computational complexity of producing a configuration may become inordinately high, and understanding how partial resolution of a configuration schema affects the rules may be difficult. We currently believe that limiting the expressions to Horn clauses which depend on compatibility and transformation information is acceptable, but this needs to be validated.

5. INITIAL EXPERIENCE USING THE MODEL FOR AVIONICS CAD

An in-house project is implementing part of the KBSA Framework CCM model to solve CCM problems in a prototype avionics design capture system. In their domain, multiple teams of designers, responsible for different subdomains, are evolving large and complex subsystem designs which must be periodically integrated to yield a design of the complete system. A typically development process, in the abstract, would look like this.

- The system administrator creates the database and defines the aircraft class which the database describes.
- The system integrators for the various subdomains define the subsystems from which data is required. Working with the system administrator, they also define user rights for the various subsystems.
- The designers within the various domains create and elaborate the design objects making up their subsystems, including intra- and inter-subdomain interfaces.
- The system integrators make sure that the data entered is consistent, complete, and on schedule for system integration releases. The system integrators also assist in any co-ordination among teams that may be required. Depending on the subdomain, revision management is the responsibility either of the subsystem designer or the subdomain's system integrator.
- A distinguished system integrator is responsible for aircraft configuration management (ie, for configuration management of the complete system).

5.1 REPRESENTING CHANGE

It was necessary to retain the revision history of these avionics parts objects - for example, the descriptions of fielded revisions must be available as long as the revisions are in service. Saving the entire database - a snapshot of the world - each time a configuration needed to be retained would require approximately 300 gigabytes per class of aircraft (approximately 100 subsystems each requiring about 30 snapshots); this was deemed undesirable.

The first-cut solution was to maintain a revision history on an object-by-object basis; an object was checked-in by marking it "immutable", and a configuration was simply a list of pointers to frozen (checked-in) objects. Any future changes to a frozen object would need to be made to a new checked-out version - a mutable copy -of that object. But the database is highly interconnected; a change to one part has effects beyond the strictly local modifications. When checking-in an object, it is necessary to also check-in the objects which are components of that object, since they are part of its definition; when checking-out an object, its components must be checked-out also. The more difficult question was how the objects which reference an object should respond to a new version of that object. Copy propagation problems arise; the objects which reference the old version of the object may need to themselves be duplicated, since a new version of the object referenced is a change to a feature of the objects which reference it. Thus it was necessary to produce not only a new version (copy) of the object to be checked-out, but also potentially of all the objects which referenced it - in the worst case, copying the entire database again.

Criteria which could be used to limit the recursive copy propagation without sacrificing correctness and consistency were not readily apparent; we contend that this will be true of most domains. The KBSA CCM model, which is conceptually clean and leads to a space-efficient representation of a tree of configurations, was therefore chosen. Configurations were implemented as growable hash tables; a successor configuration is a forward diff from its predecessor.

A requirement for this avionics domain was to facilitate the work of multiple cooperating teams via decoupling the subdomains during development, while facilitating system integration. The multiple subdomains were decoupled into multiple trees of configurations. Each subdomain maintains their configurations as a tree of state tables, where each state table records the modifications made to the database since the previous state table. Configurations from the subdomains are periodically composed to form one global system configuration, a node in the system configuration tree; the system configurations cache the complete state of the system.

5.2 DYNAMIC REFERENCES AND CHANGE NOTIFICATION

The use of dynamic version cursors supports controlled inter-domain references during development, facilitating system integration. The protocols for cross-configuration reference satisfaction and change notification are computer assisted, requiring the okay of system integrators from both domains.

If an object in subdomain X must reference an object in subdomain Y, the object's definition is not referenced directly; a dynamic reference (dynamic version cursor) is used, which may be "satisfied" by zero or more objects in the Y subdomain. Creation of a version cursor initiates the cross-configuration reference protocol, which is used to inform one domain (the donor domain, which receives the request) that an object in another domain (the recipient domain, which sends the request) wants to depend on or "point to" an object in the donor domain. The protocol guarantees that the interdomain links and back-links between objects are set only upon explicit agreement from both domains.

A list of unsatisfied version cursors is maintained, and the list must be cleared before a configuration can be saved. During the "system integration" (consistency establishment) process, the distinguished system integrator is responsible for deciding the cause of unsatisfied cursors; that individual may direct the submitting design group to remove the cursor, direct the submitting group to restate the version cursor, or direct the target design group to satisfy the cursor.

A slight complexity is introduced when the target of a previously satisfied cross-configuration reference is changed, resulting in a now-unsatisfied version cursor. The dynamic version cursor capability also supports a change notification facility which notifies a user of an object if the object has been changed such that it no longer satisfies the version cursor's selection rule. When a cursor becomes unsatisfied, the change notification protocol is activated; change notifications inform one domain (the recipient, which receives the notification) that an object in another domain (the donor, which sends the notification) has been modified so significantly that the links and back-links between objects in the recipient domain and the modified object in the donor domain should be reevaluated.

The prototype avionics design capture system has been implemented in Common LISP, and tested with small test problems; it is now being exercised with real data (15000 objects, about 10 megabytes of avionics design data).

6. RELATED ISSUES AND OPEN ISSUES

There are a variety of issues tangentially related to CCM which the KBSA framework CCM model does not yet address; the model also requires exercising and improvement.

- *Exercising the Model.* The CCM model has a solid conceptual base, but some key areas need further elaboration. Some areas have requirements, but not a notation, defined - eg, the form of environment descriptions or compatibility attributes. Some areas have unresolved issues - eg, can multiple representations for the same conceptual object be tightly associated with each other, such that they can be referenced and managed in a conceptually clean fashion? The model is currently analogous to a set of axioms and some core theorems - it needs to be further prototyped and exercised, to expand the number of theorems and to evaluate and improve the model.

- *CCM and Activities Coordination.* It is clear the CCM and the activities coordinator will be closely related. Both are concerned with managing the changes which occur to project state, and with facilitating (or automatically performing) some operations on project state while prohibiting others. The activities coordinator will play the major role in enforcing *process consistency,* to better lead to the *product consistency* which is the major concern of CCM. The activities coordinator will be a key resource in automating the CCM process, to reduce its intrusiveness as much as possible, and to extend its benefits to the full scope of both informal and formal changes. It will also play a key role in proving configurable CCM policy; an integrated CCM facility should provide the *mechanism* necessary to implement the *policy* chosen by the project.

- *Reuse.* Combined with better interface specifications, the KBSA CCM model has the ability to record compatibility information in a machine manipulable fashion. Just as early software developers advanced from having textual descriptions of how to assemble systems, to "scripts", to "Makefiles" that have a specific language to reason about the construction of systems, we believe that making compatibility information explicit will make it possible to determine if some existing piece of software will work in one's environment. This is a critical part of the software reuse problem - namely, having confidence that expending the effort to use the software will be successful. While CCM modeling will be able to determine the compatibility of a software module and make integration of the component easier, the problem of identifying what components may be applicable is outside the scope of the CCM model; the CCM model deals with issues of compatibility and recoverability, but only marginally with issues of module classification and library search.

REFERENCES

[BJORN] A. Bjornerstedt, C. Hulten, "Version Control in an Object-Oriented Architecture," in *Object-Oriented Concepts, Databases, and Applications*, W. Kim, F. Lochovsky, editors, ACM Press, 1989.

[ORION] J. Banerjee, W. Kim, N. Ballou, H. Chou, J. Garza, D. Woelk, "ORION-1 Data Model and Interface, Rev. 2", *MCC Technical Report Number: DB-093-86, Rev. 2(Q)*, March 31, 1987.

[EIS] *EIS Specification Volume I: Organization and Concepts* (CDRL 13,16,17 under WRDC contract F33615-87-C-1401), October 1989.

[TI] J. Joseph, M. Shadowens, J. Chen, C. Thompson, "Strawman Reference Model for Change Management of Objects," Texas Instruments, 1990.

[KATZ1] R. Katz, "Towards a Unified Framework for Version Modeling," University of California, Berkeley.

[KATZ2] R. Katz, *Information Management for Engineering Design*, Springer-Verlag, 1985.

[KBSAFW] A. Larson, J. Clark, J. Kimball, B. Schrag, *KBSA Framework Final Technical Report Phase 2*, Honeywell Systems and Research Center, 1990.

[INSCAPE] D. Perry, "Version Control in the Inscape Environment," *9th International Conference on Software Engineering*, IEEE, March 30, April 2, 1987.

[RUM] J. Rumbaugh, "Controlling Propagation of Operations using Attributes on Relations," *OOPSLA '88 Proceedings*, September 25-30, 1988.

[REED] D. Reed, "Implementing Atomic Actions on Decentralized Data," *ACM Transactions on Computer Systems*, Vol. 1. No. 1, February 1983.

[RCS] W. Tichy, "RCS - A System for Version Control," *Software-Practice and Experience*, Vol. 15(7), July 1985.

[Gaia] P. Vines, D. Vines, T. King, "Configuration and Change Control in Gaia," Honeywell.

Animated Knowledge-Based Requirements Traceability

Dr. Warren Moseley

Texas Instruments

P.O. 655474, M.S. 238

Dallas,Texas, 75265

214-995-0782

Abstract:

Quality improvements can be made in the software engineering process, by ensuring that product and process traceability are built into the supporting environment. The foundation for a system that will support traceability must include a mature software process model.

A new CASE(Computer Aided Software Engineering) approach must first depend on a mature software process model before the knowledge about the support environment for that model can be introduced. Software product quality is enhanced through requirements traceability - the ability to trace a module back through integration, to implementation, to design and finally to requirements. There are two types of knowledge associated with traceability, product knowledge and process knowledge. Product knowledge traces product requirements to actual code implementation, while process knowledge demonstrates the inherent structure by which the traceable links are constructed in the knowledge bases being built. A Poor Man's Case Tool(PMCT) introduces an embedded automated knowledge-based approach to requirements traceability.

Knowledge-Based Requirements Traceability

A new CASE(Computer Aided Software Engineering) approach must first depend on a mature software process model before the knowledge about the support environment for that model can be introduced. Software product quality is enhanced through requirements traceability - the ability to trace a module back through integration, to implementation, to design and finally to requirements. There are two types of knowledge associated with traceability, product knowledge and process knowledge. Product knowledge traces product requirements to actual code implementation, while process knowledge demonstrates the inherent structure by which the traceable links are constructed in the knowledge bases being built. A Poor Man's Case Tool(PMCT) introduces an embedded automated knowledge-based approach to requirements traceability.

Quality improvements can be made in the software engineering process, by ensuring that product and process traceability are built into the supporting environment. The foundation for a system that will support traceability must include a mature software process model. For the purpose of this paper, software process modeling is defined as a methodology that encompasses a representation approach, comprehensive analysis capabilities, and the capability to make predictions regarding the effects of changes to a process. Watts Humphrey[1] describes five levels of software maturity - initial, repeatable, defined, managed, and optimizing. It is important to understand this maturity to appreciate the necessity for traceability in a support environment. Approximately 87% of the companies observed in a study by the Software Engineering Institute were at the initial level in the maturity model. There were no companies at level three or above, only a few projects within given companies that achieved the defined status.

To achieve managed and optimized levels in the maturity model, utilization of these measurements to redefine and optimize the software engineering process is a necessity. While there are other factors involved in the maturing of the software process, the primary objective is to achieve a controlled and measured process for the project development foundation. The Software Assessment Procedures[2], developed by the Software Engineering Institute established guidelines to help software developers discover the current maturity level. In addition to this maturity level discovery, the software assessment process provides a communication framework for level increase.

An important task in the systems development process is to determine if the top-level software requirements are correctly represented in the final level of a delivered product. Requirements Traceability is a generic term used to refer to tracking software requirements through to final code. Requirements traceability is a method to ensure that not only is a software system correct, but that it is also complete. It demonstrates paths from requirements to code that the developer can trace in either direction. These traceability links are essential in the verification of the component in question, but are also a valuable tool in the assessment of software changes to that component. Automated requirements traceability must be an integral component of the software support environment to ensure maturity level increase.

There have been several attempts at the automation of Requirements Traceability[3456]. Traditional approaches place traceability in the domain of documentation and bookkeeping. Dorfman describes ARTS(Automated Requirements Traceability System) [7] as a bookkeeping program that operates on a database consisting of system requirements and their attributes. While the storing of the bookkeeping information is important in the traceability process, the knowledge associated with traceability needs the natural explanation mechanism of knowledge based systems. Reifer and Marciniak[8] suggest a knowledge-based approach to software life-cycle management and imply that knowledge will become a more integral part of the entire acquisition and delivery process. Figure 1 depicts the Reifer-Marciniak knowledge-based approach to life-cycle management. Carefully note the iterative nature of this approach. Each iteration emphasizes knowledge capture. Knowledge intensive requirements place a demand on the knowledge engineering aspects of the software product development.

Conventional approaches to traceability dictated a mapping from requirements to test item, and used this mapping to demonstrate conformance to the customer. A key quality issue in this process is the measure of the conformance/nonconformance to requirements. This is usually demonstrated by a requirements traceability matrix. The traceability matrix contains only product knowledge and lacks knowledge associated with the inherent chaining structure naturally found in rule-based systems[9] [10]. Rule-based systems have the ability to explain the line of reasoning invoked in the carrying out of a task. Chaining rules together in a coherent fashion provides the support for this explanation facility. The tool described in this paper invokes the natural chaining mechanism to carry out the task of traceability. The traceability is built into the linkage mechanism of the knowledge base that represents the steps necessary to produce the product.

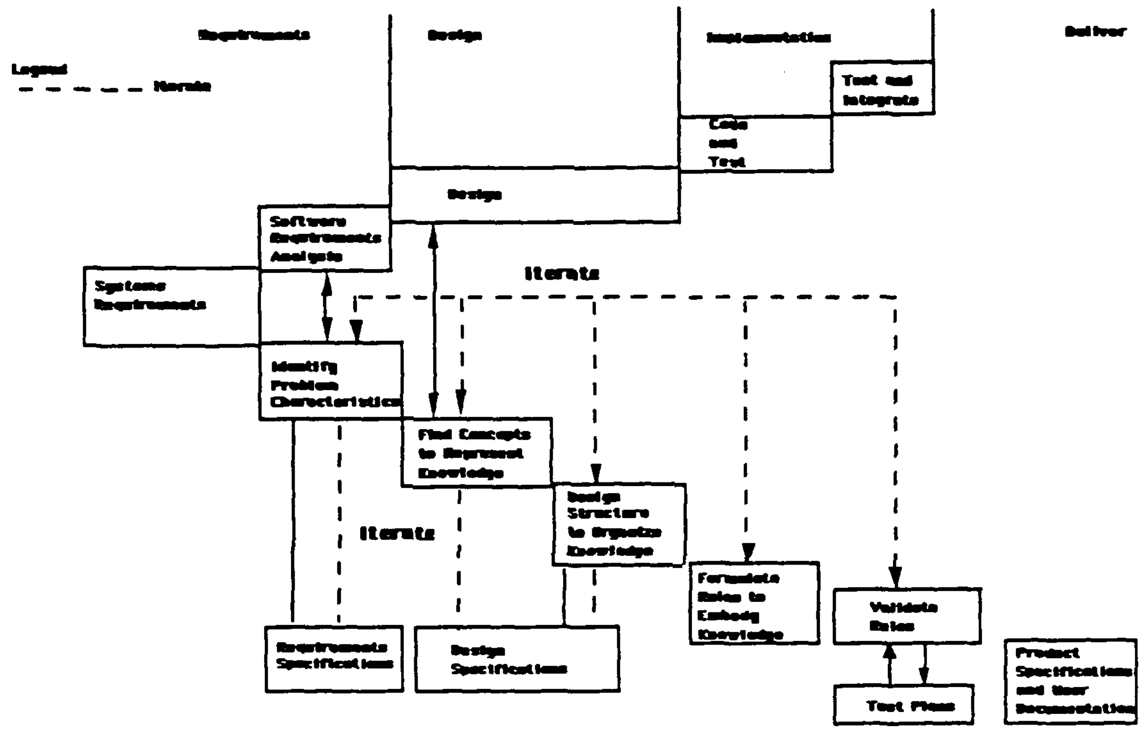

Requirements
Design
Implementation
Deliver
Legend
Iterate
Test and Integrate
Code and Test
Design
Software Requirements Analysis
Systems Requirements
Iterate
Identify Problem Characteristics
Find Concepts to Represent Knowledge
Design Structure to Organize Knowledge
Iterate
Formulate Rules to Embody Knowledge
Validate Rules
Test Plans
Requirements Specifications
Design Specifications
Product Specifications and User Documentation

Figure 12.3 Knowledge System Development in the Software Life Cycle

Figure 1
The Knowledge-Based Software Life-Cycle

The delay of requirements traceability until the latter stages of a software project will no longer be acceptable. Traceability must be a subset of the strategic knowledge involved in all levels of the software development process[11]. Traditional mappings generated by the bookkeeping / documentation approach provide no insight into the quality principles of Requirements Engineering. Standards and practices dictate approaches to translate top-level requirements into written form. Few projects establish traceability early because of the ambiguities of the written software specification. Each prose paragraph contains one requirement or several requirements. A requirement may also be referred to by several paragraphs. Even though these references are stored in electronic form, this falls short of the naturally traceable knowledge associated with requirements. The clerical burden placed on the project participants often deters the progress of the product development. Conventional approaches demonstrated that the traceability was an added burden. Traceability integrates the way in which the designer thinks with the knowledge associated with the development process. It is not a mechanism that is used just to show customer satisfaction.

Computer-Aided Software Engineering research has placed too much emphasis on trying to provide tools to support the bookkeeping of software life-cycle management, only to find that the life-cycle had no mature software process to give it foundation. It is important to carefully distinguish between the idea of process and life-cycle. A process will be thought of as an ongoing activity, where a life-cycle will have specific beginning and ending tasks. Product traceability belongs to the life-cycle aspects of the project, but the conceptual foundation of traceability should be an integral part of the knowledge associated with, and captured by, the process and the product. This inherent linking of pieces of an analyzed process into a synthesized solution transcends a one-time application of the software process. In the software maturity model[12] it is crucial that there be measured improvements in the software process.

Active knowledge bases with visual representation provide a repository for software requirements in the PMCT. Building these knowledge bases visually represents the software specification preparation process. PMCT produces documentation as a by-product of the knowledge capture process. The specification for a system is an active component in the knowledge base that enhances the design process. The inclusion of traceability will be implicit in the construction of the knowledge bases. This knowledge base is executable and the traceability is provided through the explanation functions of the expert system tool selected.

There are typically two types of questions that an acceptable expert systems should be able to answer about the reasoning process. Note that these questions are not answers to a query of a project database. These questions are:

Why did you arrive at this conclusion?

How did you derive such an answer?

These also are two important issues(questions) in the conceptual framework of traceability:

1. **Why** is this component necessary to confirm this requirement?

2. **How** does this component trace to its requirement?

Figure 2 shows a Computer Systems Configuration Item(CSCI) in a real-time embedded system, created with the PMCT's Visual Object Manager. This will serve to demonstrate the use of a knowledge-based approach to managing requirements.

This example contains three major requirements which decompose into four designs which decompose into eight high-level modules. The knowledge base provides a proper framework for configuration control, and this knowledge plays an integral part in the product development . Each CSCI and the components of the CSCI are defined, placed in the reusable repository, and put into the active knowledge base which reflects the product to be delivered. As each CSCI is deemed necessary, it is entered into the knowledge base. This would provide the expert system shell with a goal or hypothesis. The goal of the above knowledge base would be to prove that CSCI 15 fully meets the logical, organizational, and physical criteria to make CSCI 15 true. As each requirement was added to the functionality of CSCI 15, an entry would be made into the knowledge base.

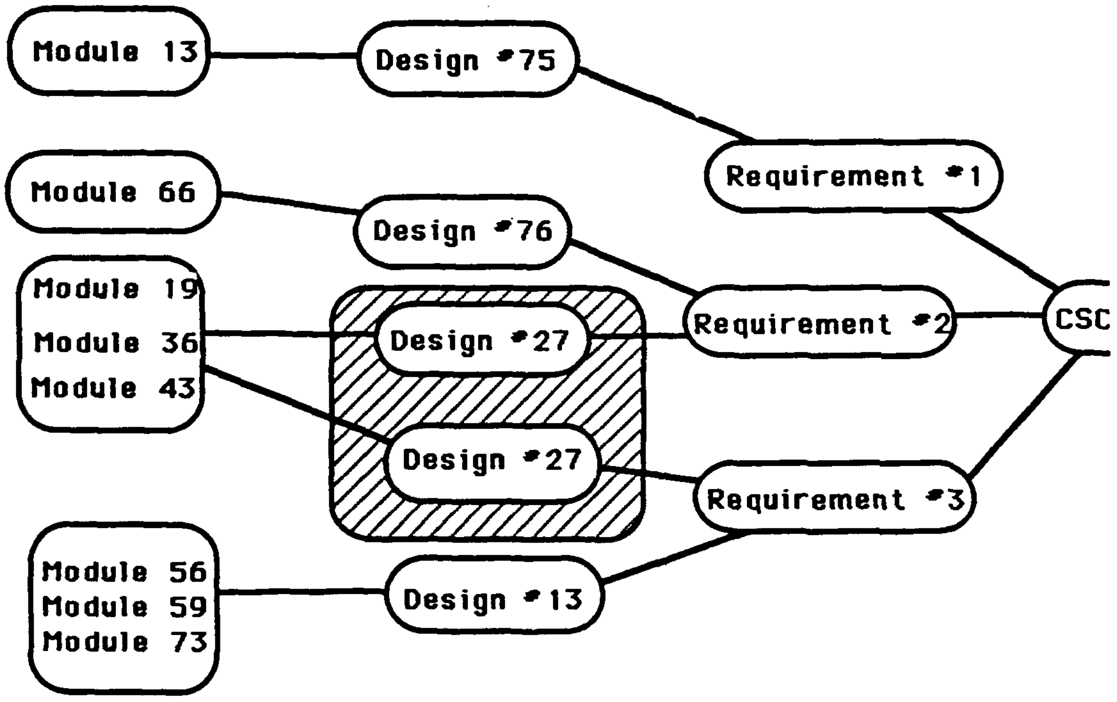


Figure 2
A Knowledge Base Representation of a Computer System Configuration Item

The diagram in Figure 3 only represents the structure of the knowledge base and not some of the actual contents of this knowledge base. The structure does not reflect any criteria except the customer explicit requirements that are called for in the Request for Proposal or the proposal negotiations.

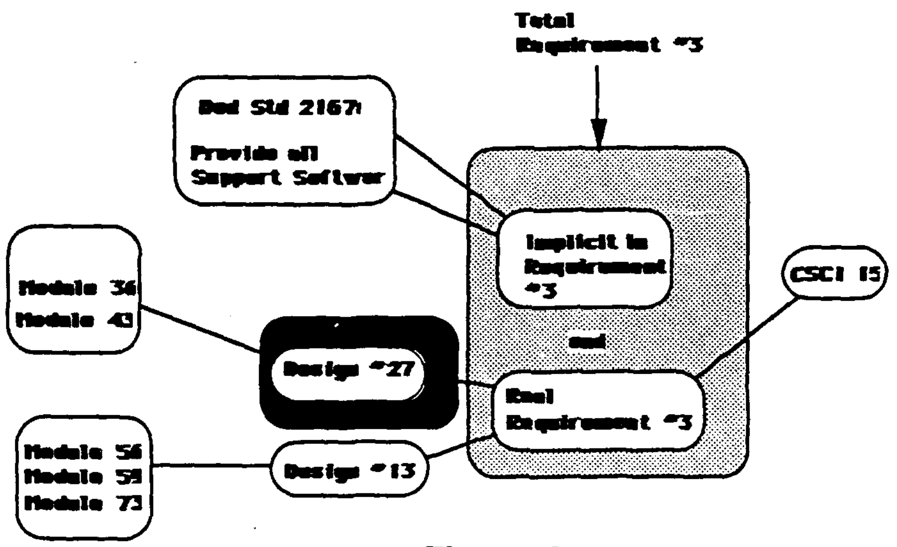


Figure 3
Implicit Requirements
must be traced

In the light shaded area in the above figure is another important aspect of the concept of applying a knowledge base to the software specification representation. The only requirements in the product documentation is the real requirement #3. However it is important in meeting contractual obligations that there are written requirements but also other types of requirements. This problem is described in Dorfman[13] quite well. Other types of requirements are standards that must be met, support software that must be delivered, specific grades of personnel that must be assigned, etc. All of this knowledge is added into the knowledge base, before requirement #3 becomes complete. In conventional approaches[14][15][16] the inclusion of different types of knowledge is often awkward, if not impossible.

There are two basic chaining mechanisms that are important in the building of production rule knowledge based systems. These are backward chaining and forward chaining. Backward chaining starts with a goal and tries to determine if all of the intermediate goals and premises of the goal are true. It searches the "then" part of the knowledge tree for a "then" clause that would match the overall goal. In our example CSCI 15 would be the goal and the three requirements necessary to satisfy that goal would be the intermediate goals. These intermediate goals are necessary to prove that goal true. Figure 4 shows the order of execution of the search of the knowledge base using backward chaining by attaching numbers to each node in the knowledge base.

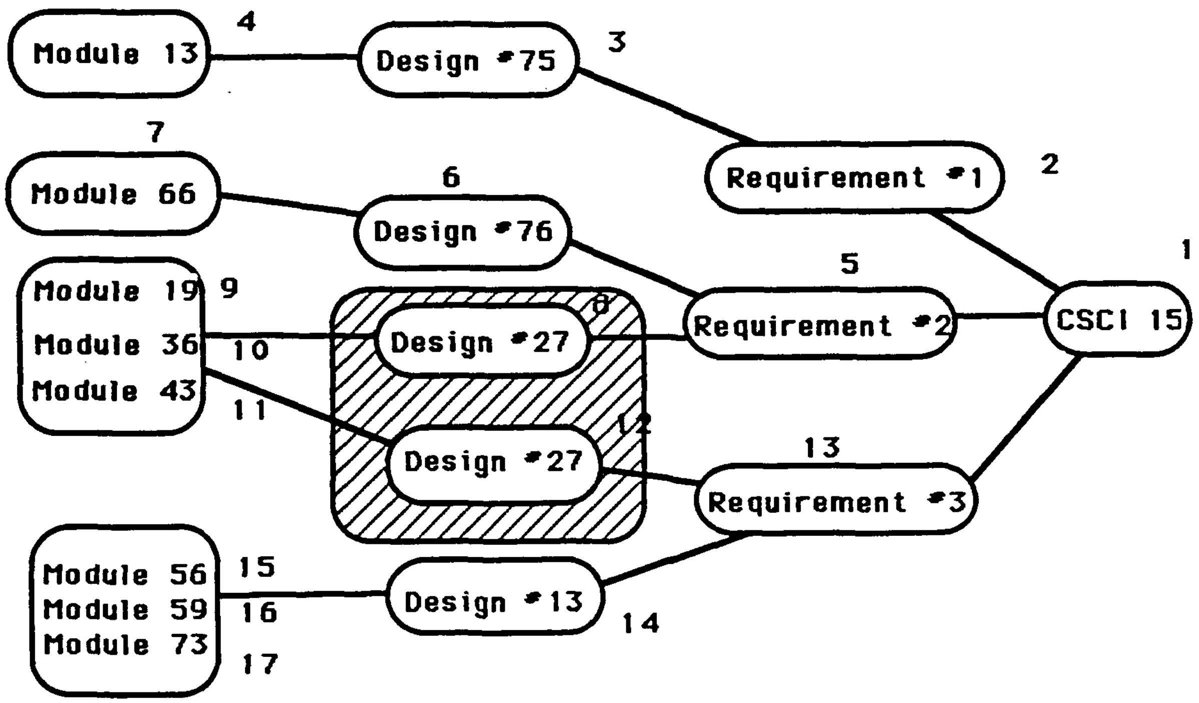


Figure 4
The Knowledge Base Trace

In determining the traceability of components of a system, one would use backward chaining, because it is important to verify that the high level hypothesis is true. The reasoning mechanism would assume that CSCI 15 was true and proceed by the numbers to try and verify or disprove that hypothesis. The three requirements are now intermediate goals that must be true for the final CSCI 15 to be met. This is quite a simplistic example for illustrative purposes, and does not have a situation in which requirements are interdependent, but there is no reason why such a structure cannot exist.

As each requirement is addressed in the design processes, the knowledge base is expanded. The concept of explanation[17] in knowledge-based systems is one of the unique features of such systems. Consider Figure 5.

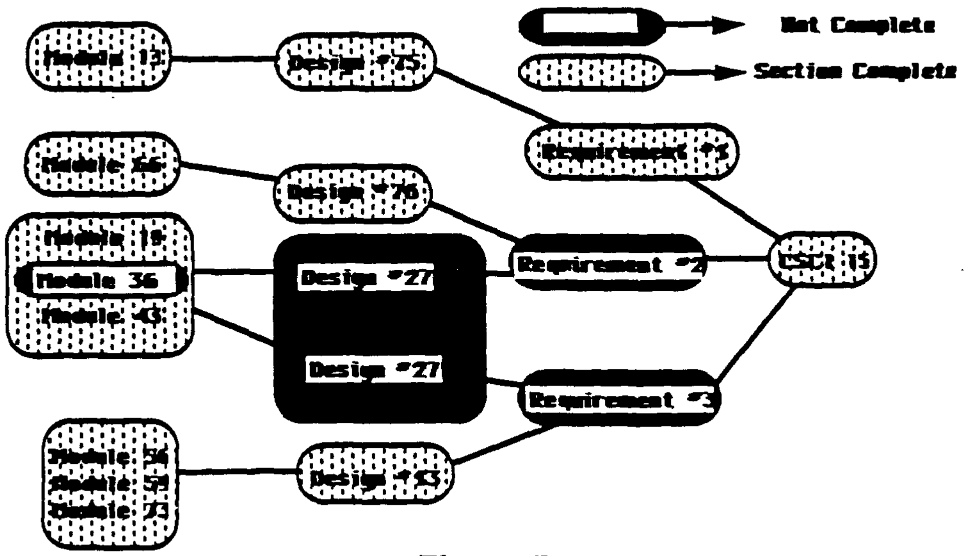


Figure 5
Complete vs Non-Complete Requirements

The areas with black background indicate that a section is still in preparation. Questioning the knowledge-based system as to the status of CSCI 15, it would reply that CSCI 15 is not met. Further inquiry using the Why option of the explanation facility would explain that Requirement #1 has been met, but that Design #27 is still not satisfactory to complete Requirement #2 or Requirement #3. All of the components of Design #27 are complete except Module #36. If the system was asked, "How did it arrive at this conclusion?", it would explain that Module #36 was necessary to complete Design #27. Design #27 is necessary to complete Requirement #3 and Requirement #2. Evidence needed to arrive a true hypothesis for CSCI 15 consists of successful completion of all of the nodes in the knowledge tree. If this was a complex system, there could be hundreds of cases that could possibly contain this scenario. It is very difficult to reason about that kind of information in the bookkeeping style of a requirements traceability matrix, even with the support of the project databases[18][19][20] in more automated scenarios. By capturing the requirements, design and implementation as premises in a knowledge base, the specification becomes an active component in the system and not just a passive part of some boring documentation. The documentation stays current because it is an active part of the design process.

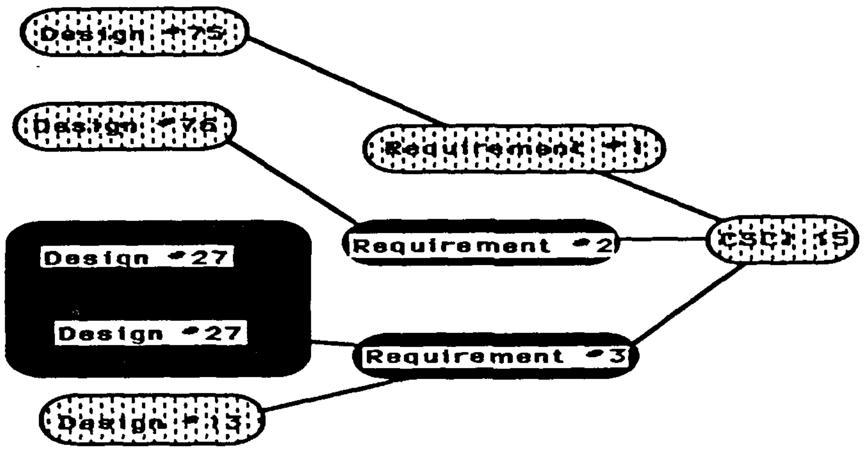


Figure 6
Initial View of Project

The knowledge base approach is also helpful from a different perspective. The project manager or software engineer can use the knowledge base as a Project Planning tool. Suppose that the structure above was the finalized structure that accurately represented the development of CSCI 15. In the early stages of the construction of the knowledge base, it might not be so clear as to the evolving structure. At this point the manager can use the knowledge base along with some available tools such as spreadsheets, pert charts, and other planning tools to come up with a more accurate estimate. In the early stage of the project, management will develop a forecast of the timeline and the people power involved. At this time the knowledge base can be used as a simulation tool. The project manager could construct an imaginary form of the project knowledge base. Assume that the project was envisioned to be of the structure found in Figure 6. At this point in the project the project manager would use this structure to help prepare the software [illegible]. There could be many anticipated scenarios involved before the final structure becomes a reality.

An important step in the project implementation is that of risk control. Figure 7 is a sample of the Spiral Model of Software Development as proposed by Dr. Barry Boehm.[21] This process model starts just to the left of the center on the west pointing axis. Each time the spiral proceeds across the north pointing axis the next step is risk analysis followed by prototyping. Each turn of the spiral, the risk analysis becomes more and more critical. Traceability has not traditionally been considered a part of the calculation of the risk associated with the turning of the spiral. However if the concept of traceability was an integral part of the preparation of these risk analyses, and this knowledge associated with the risk was attached to the executable traceable knowledge base, then the evaluation of risk becomes manageable, and more a part of the preparation of the requirements, design and implementation of the software product.

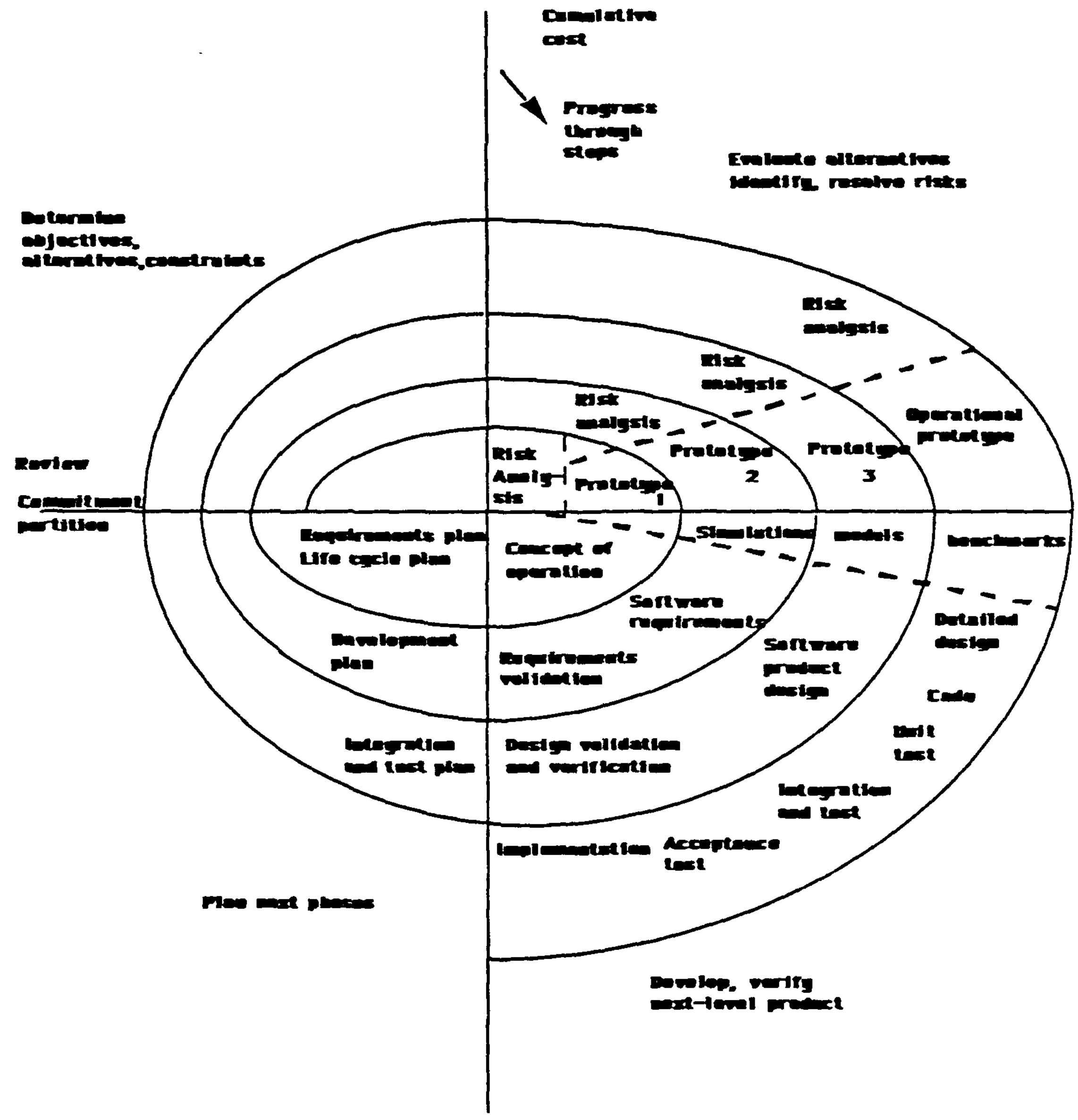


Figure 2.18 Spiral Model of the Software Process

Figure 7
Spiral Model

Expert System Shells manage the uncertainty of dealing with the domain specific knowledge bases. Certainty Factors attached to the condition-action sections of a knowledge base represents the confidence that the user has in a piece of evidence. There are numerous ways of representing certainty factors. Assume the use of uncertainty factor such as the uncertainty factors of Emycin[22]. The rule in figure 8 reads if requirement #3 is true with certainty of 99% and requirement #1 is true with certainty of 60% the one can conclude that CSCI15 is true.

If Requirement 3 [cf = 99] and
Requirement 1 [cf= 60]
then CSCI15

Figure 8
Emycin Style Rule

Often it is difficult to communicate uncertainty in the software development plan. By using a knowledge base that allows for manipulation of the uncertainty factors, this presents a help feature for the manager and software engineer to provide a common ground for communication of issues that are related to risk.

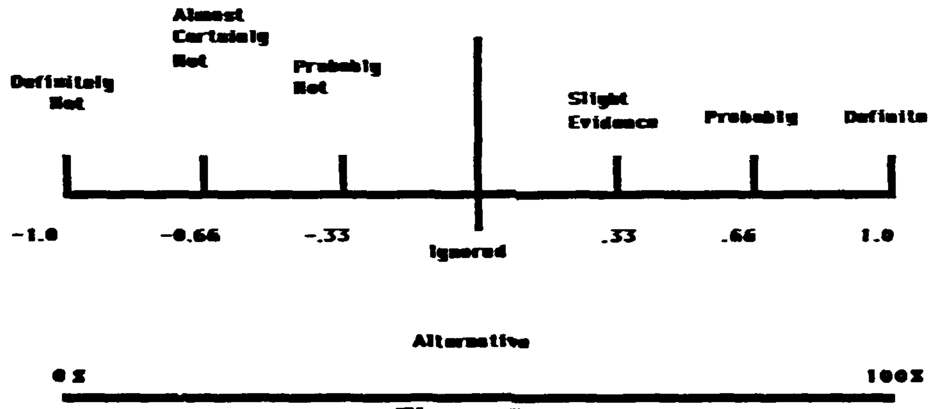


Figure 9
Two Alternative Certainty Factor Scales

In Figu 10 the manager has a high degree of certainty that requirement #1 and requirement #2 can be met. Although it is originally conceived that reuse of design #27 would facilitate the implementation of requirement #3, there is still problem with design 13. This could be a critical timing problem, or perhaps the technology at proposal time did not even exist. In the execution of a large military system, the life-span can sometimes be as much as 20 years and at conception time the technology might not even exist, implying that the user might be counting on a technology innovation to occur. This presents a high risk endeavor, and should be treated with caution.

Execution with uncertainty and explanation capability allows the Project Manager, Systems Engineer, and Software Engineer simulation capability. With a knowledge base the project manager, or software engineer can simulate the execution the Computer Systems Configuration Item. By considering different scenarios the project manager and lead engineer can adjust the certainty factors so that the overall project risk can be reduced. In this paper only simple microcomputer based tools were used, but in large complex procurements, the knowledge base could be much more robust and include other types of knowledge representation other than production rules.

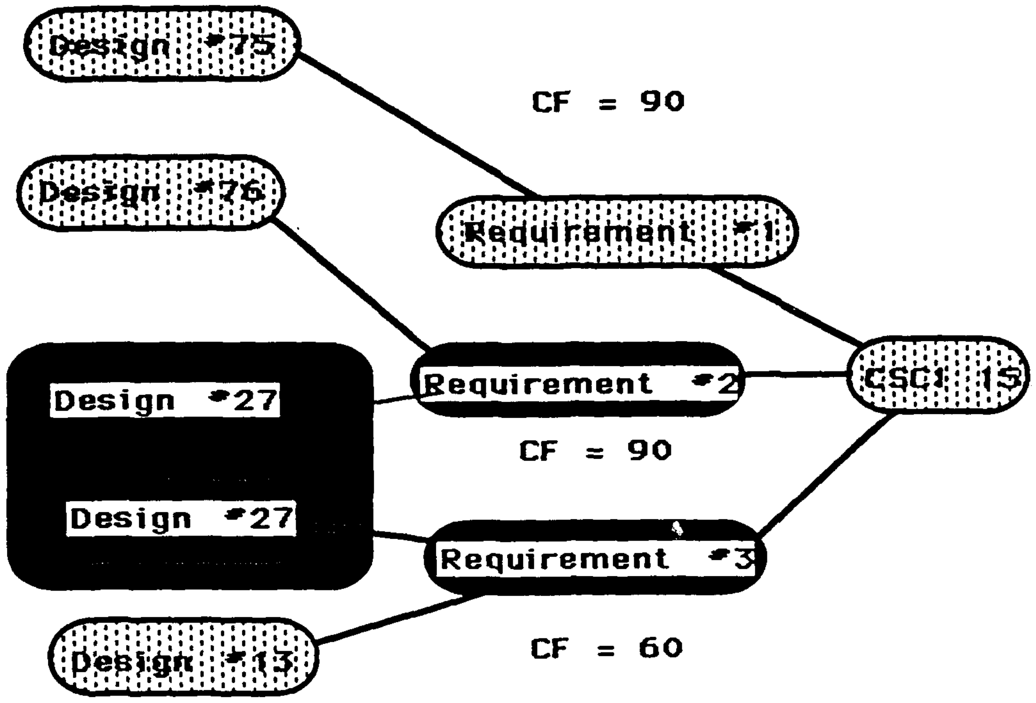


Figure 10
Attachment of Certainty Factors
to Traceability Knowledge Base

Summary

This paper focuses on a new tool that demonstrates knowledge capture for requirements traceability. Both product knowledge and process knowledge are necessary for the factors to be in place to assure increased quality in the overall software process.

[1]Humphrey, Watts, Managing The Software Process, Addison Wesley, 1989.
[2]A Software Assessment for Government Contractors
[3]Dorfman, M, and Flynn, M. ARTS- An Automated Requirements Traceability System, The Journal of Systems and Software, Vol. 4,No. 1, pp 63-74, 1984.
[4]Sciortino, J, Dunning, D., Proceedings of the AIAA/IEEE 6th Digital Avionics Systems Conference. Baltimore, Md. 1985.
[5]Pirnia, S.,Hayek, M., NAECON 1981. Proceedings. pages 389-394.
[6]LaGrone, D., Wallach E, Requirements Traceability using DSSD,Tooling up for the Software Factory, Feedback 86, Topeka, KS.
[7]Dorfman, M, and Flynn, M. ARTS- An Automated Requirements Traceability System, The Journal of Systems and Software, Vol. 4,No. 1, pp 63-74, 1984.
[8]Reifer, D, Marciniak,J. Software Acquisition Management, John Wiley Press To Be Published., 1990.
[9]Harmon, Paul, King, S. Expert Systems: Artificial Intelligence in Business, Wiley Press, NY., 1985
[10]Seigel, Paul, Expert Systems: A Non-Programmers Guide to Development and Applications, TAB Books, 1986,Summit Pa.
[11]DOD-STD-2167a, Military Standard, Defense System Software Development, MCCR. Software Standard for Mission Critical Applications
[12]Humphrey, Watts, Managing The Software Process, Addison Wesley, 1989.
[13]Dorfman, M, and Flynn, M. ARTS- An Automated Requirements Traceability System, The Journal of Systems and Software, Vol. 4,No. 1, pp 63-74, 1984.
[14]Dorfman, M, and Flynn, M. ARTS- An Automated Requirements Traceability System, The Journal of Systems and Software, Vol. 4,No. 1, pp 63-74, 1984.
[15]*Sciortino, J, Dunning, D., Proceedings of the* AIAA/IEEE 6th Digital Avionics Systems Conference. Baltimore, Md. 1985.
[16]LaGrone, D., Wallach E, Requirements Traceability using DSSD,Tooling up for the Software Factory, Feedback 86, Topeka, KS.
[17]Harmon, P, King, D, Expert System, Artificial Intelligence in Business, John Wiley Perss, New York, 1985, page 16.
[18]Dorfman, M, and Flynn, M. ARTS- An Automated Requirements Traceability System, The Journal of Systems and Software, Vol. 4,No. 1, pp 63-74, 1984.
[19]Sciortino, J, Dunning, D., Proceedings of the AIAA/IEEE 6th Digital Avionics Systems Conference. Baltimore, Md. 1985.
[20]LaGrone, D., Wallach E, Requirements Traceability using DSSD,Tooling up for the Software Factory, Feedback 86, Topeka, KS.
[21]Boehm, B., A Spiral Model of Software Development and Enhancement, IEEE Transactions on Software Engineering, May 1988, p 61-72.
[22]Buchanan, B. and Shortlife, E. Rule-Based Expert Systems: The Stanford Experiment

A Comparative Assessment of Formal Specification Techniques*

Algirdas Avižienis and Chi-Sharn Wu
Dependable Computing and Fault-Tolerant Systems Laboratory
Computer Science Department, UCLA
Los Angeles, CA 90024, U.S.A.
Tel: (213)825-3028

Abstract

A classification for formal specification techniques, which is helpful for a comparative survey, is presented. In the classification, formal specification techniques are classified into three different approaches : *operational*, *definitional* and *hybrid*. Depending on either data abstraction or sequencing is emphasized, both the operational and definitional approaches can be further split into two schools : *data paradigm* and *process paradigm*. Thus five categories in the classification are identified, and then some representative formal specification techniques in each category are briefly surveyed. Finally, a comparative assessment over these specification techniques is given based on a set of criteria, such as applicability, useability, verifiability, etc.

1 INTRODUCTION

A complete specification system consists of *methods*, *languages* and *tools*. Specification languages can be classified into three groups according to the level of formality : *informal*, *semi-formal*, and *formal*. Informal specification languages, mainly referring to natural languages, can contain many deficiencies, like inconsistency and ambiguity, which are difficult to detect. Semi-formal languages have well-defined syntax and partially defined semantics which make building automatic tools possible. SA [DeM78], SREM [Alf85], and PSL/PSA [TH77] are some of the well-known semi-formal specification systems; these systems are widely used in industry because documents resemble those written in natural language and the semi-formal languages can be learned and understood with limited effort by people who did not have extensive training in formal methods. Formal specification languages, with well-defined syntax and semantics, have the advantage of being concise and unambiguous; they support formal reasoning about the functional specification, and provide a basis for verification of the resulting software product.

The objective of this paper is to have a survey on some representative formal specification techniques and make an assessment of them. A classification for formal specification techniques is presented in Section 2. In section 3, some formal specification techniques are surveyed in the order according to the classification. In section 4, a comparative assessment over the surveyed specification techniques is given based on a set of criteria.

*Supported by Calspan UB Research Center, Subcontract No. C/UB-24, "Methodology and Languages for System Specification".

Operational Approach :
(using VDM notation)

$init : \rightarrow \text{seq of } ELEM$

$init() \quad \triangleq \quad [\,]$

$push : \text{seq of } ELEM \times ELEM \rightarrow \text{seq of } ELEM$

$push(stk, elem) \quad \triangleq \quad elem \frown stk$

$pop \;\; (stk: \text{seq of } ELEM) \; stk': \text{seq of } ELEM$
$\text{pre } stk \neq [\,]$
$\text{post } stk' = \text{tl } stk$

Definitional Approach :
(using equational axioms)

init : → STACK

push : STACK, ELEM → STACK

pop : STACK → STACK

pop(init) = UNDEFINED

pop(push(stk,elem)) = stk

(a) Data Paradigm

Operational Approach :
(using CSP notation)

$STACK = P_{<>}$
$where \; P_{<>} = push?x \rightarrow P_{<x>}$
$\qquad P_{<x>\wedge s} = (pop!x \rightarrow P_s \mid push?y \rightarrow P_{<y>\wedge<x>\wedge s}$

(b) Process Paradigm

Figure 1: Different formal specifications for the behavior of stack

2 CLASSIFICATION

We can categorize the formal specification techniques into three different approaches : *operational*, *definitional* and *hybrid*. Using the operational approach, a system is described as an abstract model by which the behavioral properties exhibited are those desired for the specified system. Using the definitional approach, systems are specified by such behavioral properties directly. In a hybrid approach, a specification method is extended by combining with other formalism for specifying more kinds of properties. In Figure 1(a), the fundamental difference between the operational and definitional approaches is illustrated by showing how to specify the behavior of stack. In operational approach, a predefined data type, **sequence**, is used to model the stack : *push* is modeled by concatenating an element to the head of sequence, and *pop* is modeled by deleting the head element (using tail function) with the pre-condition that the stack is not empty. In definitional approach, however, no explicit model (or data structure) is used; the behavior of stack is specified by two equational axioms : the first one states that the empty stack cannot be popped, and the second one describes the last-in-first-out property. In the literature, researchers also call the operational approach *constructive* and the definitional approach *axiomatic*.

In [Jac88], specification techniques are classified into *data paradigm* and *process paradigm*. Based on that viewpoint, both the operational and definitional approaches can be further split into two schools : *data* school, which advocates the primacy of data abstractions, and *process* school, which focuses on sequences of events or actions (operations). That is, the prime concerns of the approaches based on the data and process paradigms are *data* and *sequencing* respectively. The two examples in Figure 1(a) use the data paradigm. An example of process paradigm is shown in Figure 1(b)

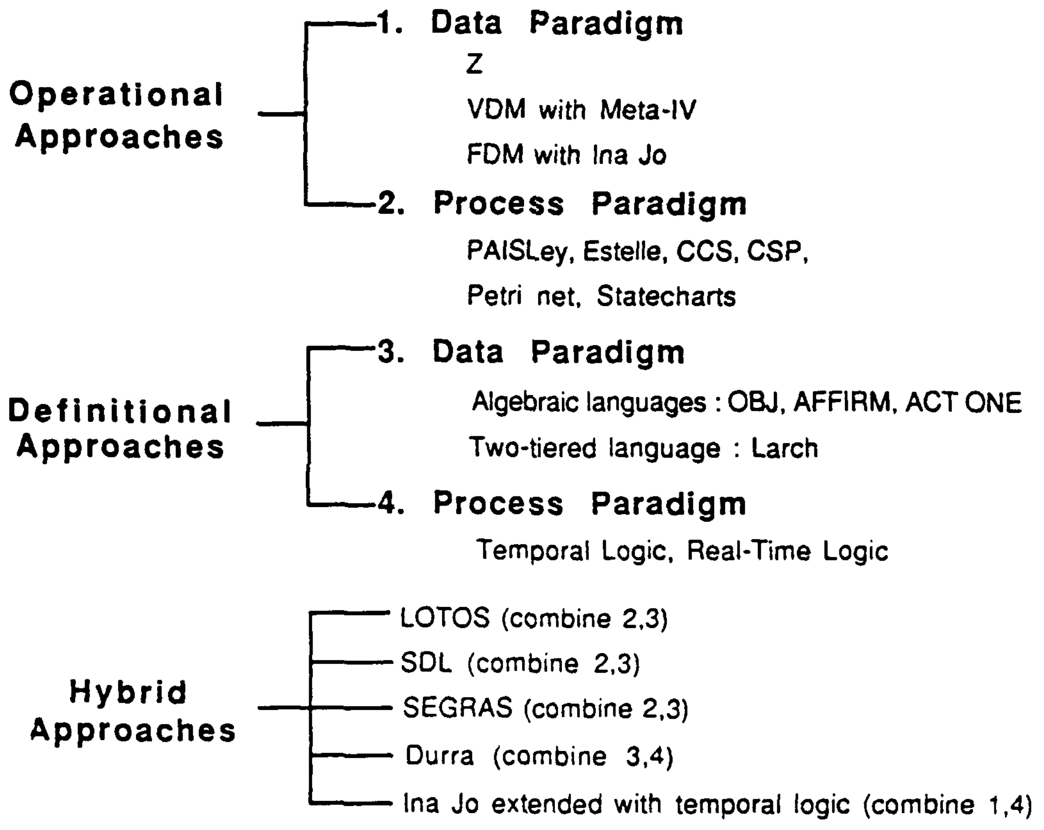


Figure 2: *Classification for some formal specification techniques*

where CSP notation (an operational approach) [Hoa85] is used; the specification states that (1) the stack is empty initially; (2) when the stack is empty, it is ready to engage input event *push* for getting an element; and (3) when not empty, either *pop* (output event) or *push* can be engaged. Note that the sequencing of events is emphasized when using the process paradigm, but is implicit when using the data paradigm.

Figure 2 shows the classification for some formal specification techniques. Note that there is no way to formalize this classification since the distinction between different categories is not clear-cut. Sometimes for achieving a higher level of abstraction, the behavior of a model, although using the technique classified as operational approach, can be described by stating its properties. Also, most techniques incorporate both process and data paradigms to some degree since no practical technique can rely purely on data or process notions. However, we believe that the effort to create such a classification is worthwhile for the purpose of a comparative survey.

In general, both data and process paradigms are marred by their biases. The data approaches do not handle concurrency well. The lack of data abstraction in the process approaches creates complexity and inflexibility to changes in data representation. So in [Jac88], Jackson contends that each school has much to offer, and that an effective approach to software development must contain ingredients from both schools, data and process, in a reasonable balance. As shown in Figure 2, it is interesting to note that each hybrid approach indicated there is based on combining the techniques of both data paradigm and process paradigm.

3 SURVEY OF FORMAL SPECIFICATION TECHNIQUES

3.1 Operational Approaches

3.1.1 Data Paradigm – VDM, FDM and Z

VDM (Vienna Development Method) [Jon89] was developed at IBM Vienna Research Laboratories during the 1970s. **FDM** (Formal Development Methodology) [Ber87] was developed by System Development Corporation (SDC), Santa Monica, California. **Z** [Hay87] was developed by the Programming Research Group of the University of Oxford. In the literature, they are often referred as *model-oriented* specification mehods, i.e., they rely on formulating a model of the system which defines a mathematical model[1] of its *data* and also corresponding *operations* (*transforms* in FDM term) on the data. The main guideline in constructing a model-oriented specification is to formalize data in such a way that the operations can be written in a straightforward manner.

Z and the specification languages of VDM and FDM, named `Meta-IV` and `Ina Jo`, respectively, are all based on first-order predicate logic. Meta-IV and Z provide a richer set of primitives than Ina Jo does for system modeling. In the development of Z, great emphasis on the readability of specifications has led to the development of the 'schema', a device for organizing the presentation of Z specification, which is essentially a syntactic unit for expressing part of a specification. Both VDM and FDM lack an equivalent to the schema notation of Z, but explicitly separate the pre-conditions and post-conditions in the description of operations.

3.1.2 Process Paradigm – PAISLey and Estelle

In this category, two kinds of representation techniques can be distinguished :

- *text-based techniques*, such as PAISLey [ZS86], Estelle [D+89], CCS [Mil80] and CSP [Hoa85].
- *graphics-based techniques*, such as Statecharts [Har87] and Petri net-based methods.

All the techniques use only some primitive data types, similar to mcst programming languages. for data domain description. In this paper, only PAISLey and Estelle are discussed.

PAISLey [ZS86], developed at AT&T Bell Labs for describing embedded systems, is a Specification Language which is Process-oriented, Applicative, and Interpretable (executable). It is actually based on two computational models : *functional programming* and *asynchronously interacting concurrent processes.* A system specified in PAISLey consists of a set of asynchronous processes; some processes represent virtual objects within the proposed system, while others may be digital simulations of objects in its environment. Each PAISLey process has a state and goes through a never-ending sequence of a discrete state changes, and the state changes are defined in a functional style. A mechanism, called exchange functions, is provided as a powerful means of specifying asynchronous interactions.

Estelle [DV89, D+89], is an ISO standard FDT (Formal Description Technique) for describing ISO protocols. In Estelle, a distributed system is specified as a hierarchy of communicating modules, and a module's behavior is described as a nondeterministic, communicating, extended finite state machine using Estelle primitives and Pascal statements. Estelle is the result of a compromise

[1] Data domain is modeled using well understood mathematical entities, such as set and sequence

between using high-level constructs for the formal description and being able to easily and efficiently implement those constructs.

3.2 Definitional Approaches

3.2.1 Data Paradigm – OBJ and Larch

OBJ [Gog84], developed at UCLA and SRI International, is designed to support parameterized programming, using algebraic specification technique. Based on *equational logic*, OBJ can be interpreted directly as rewrite rules; several executable versions of OBJ (interpreter) have been implemented, and OBJ3 [GW88] is the latest version developed at SRI. OBJ provides features to assist the development of correct specifications: 'objects' which allow large specifications to be broken down into 'mind-size' pieces, facilities for testing the pieces and their interconnections by executing test cases, strong typing, the systematic use of error conditions (factored out from normal behavior) and semantic consistency checks.

Larch [GHW85], developed at MIT and DEC, is a two-tiered specification technique since a Larch specification is written in two parts using different languages and logics : the Larch *Shared* Language is used to specify the abstract data types which constitute the lower tier; a Larch *interface* language is used to describe the observable behavior of program modules based on the abstract data types. Essentially, Shared Language uses algebraic technique to describe fundamental abstractions, and interface languages, which are dependent of programming languages, use first-order predicate logic to describe state transformations. In this way, Shared Language components can be reused by different interface language components, and programming language dependent issues, such as side effects and error handling, can be isolated into the interface language components.

3.2.2 Process Paradigm – Temporal Logic and Real-Time Logic

Temporal logic (TL) [Gal87], a formal language for expressing temporal properties, provides a natural way of describing and reasoning safety properties and liveness properties of a system. A structure of states (e.g. a sequence or tree of states), generated by every individual run of a program, is the key concept that makes temporal logic suitable for program specification. A temporal axiom is an assertion about state sequences, using temporal operators such as $\Box$ (henceforth), $\Diamond$ (eventually). A temporal logic specification, consisting of a set of temporal axioms, specifies properties that must be true of all state sequences resulting from system execution. Several variants of temporal logics, including different types of temporal semantics and different ways of real time extensions, have been studied by logicians and computer scientists.

Real-time logic (RTL) [JM86] is a formal language designed for reasoning about timing properties of real-time systems, especially for safety analysis. In contrast to temporal logic, RTL is intended to describe systems for which the absolute timing of events and not only their relative ordering is important. Time is captured by the *occurrence function*; the notation $@(e, i)$ is used to denote the time of the ith occurrence of event e. RTL formulas, which represent assertion over occurrence functions, are constructed using first-order logic. Given the timing specification of a system and a safety assertion to be analyzed, both in RTL formulas, the goal is to relate the safety assertion to the system specification. If the safety assertion is a *theorem* derivable from the specification, then the system is safe with respect to the behavior denoted by the safety assertion, as long as the implementation is faithful to the specification.

3.3 Hybrid Approaches

LOTOS (Language of Temporal Ordering Specification) [DV89, EVD89] is one of the two FDTs developed within ISO for the formal specification of open distributed systems. A LOTOS specification contains two components : the description of process behaviors and interactions (*process abstraction*), and the description of data structure and value expressions (*abstract data types*). Process abstraction is based on many ideas from CSP and CCS, and abstract data types are described by an algebraic specification language based on ACT ONE [EM85].

SDL (Specification and Description Language) [BH89, SSR89], developed and standardized by CCITT, has been developed for use in telecommunication systems including data communication, but actually it can be used in the real time and interactive systems. SDL has two paradigms : Abstract Data Type (described in ACT ONE) and Finite State Machines (for modeling system dynamic behavior). The user friendliness of SDL is partly due to the graphical representation, SDL/GR, in which *graphical syntax* is used to give overview. SDL/GR is complemented by SDL/PR, a textual phrase representation using only *textual syntax*, since graphical symbols are missing (being unsuitable) for some concepts, e.g. abstract data type.

SEGRAS[Kra87], a formal language for writing and analyzing specifications of distributed software systems, unifies algebraic specifications of abstract data types with high-level Petri net specifications of nonsequential systems in a common syntactic and semantic framework. The data structure of a system, the information content of its local states, and static constraints to state changes are specified algebraically using positive conditional equations. Dynamic behavior is specified by high-level Petri nets.

Durra [BW87], intended for real-time applications, is a specification language which combines two formalisms : Larch used to specify functional behavior, and an event expression language used to specify timing behavior[2]. In [WN89], Ina Jo is extended with temporal logic to specify concurrency properties; this method is referred as "Ina Jo + TL" in the following discussion.

A growing field in software engineering, called *multiparadigm programming*, has been advocated for building systems using as many paradigms as we need, each paradigm handling those aspects of the system for which it is best suited [Hal86, Zav89]. With the same spirit, applying different specification languages for different parts of a complex system in forming a composite specification is also considered as a promising way.

3.4 Summary of the Surveyed Techniques

Figure 3 shows the summary of surveyed specification techniques. Some observations are made as follows :

1. All the techniques using data paradigm, shown in Figure 3(a), are designed for specifying sequential systems. As shown in Figure 3(b) and (c), techniques using process paradigm or hybrid approach can specify either (non-realtime) distributed systems or real-time systems.

2. Z, VDM and FDM have the capability for wide range of abstraction, i.e., the languages provide constructs for specifying systems in a wide spectrum ranging from the most abstract level to the concrete level which is closely akin to the final implementation. Based on such

[2]Real-time logic is used to define the semantics of the timing behaviors.

	Operational Approaches			*Definitional Approaches*	
	Z	VDM	FDM	OBJ	Larch
Formal foundation	set theory, state machine, first-order logic	set theory, state machine, first-order logic	state machine, first-order logic	algebraic ADT	algebraic ADT, first-order logic
Application area	sequential systems	sequential systems	sequential systems	sequential systems	sequential systems
Abstraction capability	high – low	high – low	high – low	high	high
Representation style	descriptive, prescriptive	descriptive, prescriptive	descriptive, prescriptive	descriptive	descriptive

(a) Data Paradigm

	Operational Approaches		*Definitional Approaches*	
	PAISLey	Estelle	Temporal Logic	RTL
Formal foundation	async process, functional programming	async process, extended FSM	modal logic	first-order logic, event occurrence
Application area	real-time systems	distributed systems	concurrent systems	timing analysis of real-time systems
Abstraction capability	medium	medium – low	high	high
Representation style	prescriptive	prescriptive	descriptive	descriptive

(b) Process Paradigm

	LOTOS	SDL	SEGRAS	Durra	Ina Jo + TL
Formal foundation	sync process, process algebras, algebraic ADT	async process, extended FSM, algebraic ADT	High-level Petri nets, algebraic ADT	Larch, RTL	Ina Jo, temporal logic
Application area	distributed systems	telecommunication, real-time systems	distributed systems	real-time systems	concurrent systems
Abstraction capability	high – medium	high – low	high – medium	high	high – low
Representation style	prescriptive, descriptive	prescriptive, descriptive	prescriptive, descriptive	descriptive	prescriptive, descriptive

(c) Hybrid Approaches

1. ADT stands for abstract data type.
2. 'async' and 'sync' are short for 'asynchronous' and 'synchronous', respectively.

Figure 3: Summary of surveyed formal specification techniques

capability, stepwise refinement techniques for deriving implementation were developed. On the other hand, definitional approaches, such as OBJ, Larch, temporal logic and RTL, provide only constructs of high level of abstraction.

3. The representation style of specifications in Z, VDM and FDM can be either *prescriptive* (specify "how") or *descriptive* (specify "what"), depending on the desired level of abstraction and specifier's intention. The descriptive style is more abstract, and is less bound to implementational bias than the prescriptive style. The capability of using both styles also contributes to the wide range of abstraction. LOTOS, SDL and SEGRAS, on the other hand, specify dynamic behavior of the system in prescriptive style, and specify data domain, using algebraic techniques, in descriptive style.

4 A COMPARATIVE ASSESSMENT

In this section, the surveyed specification techniques are assessed based on the following criteria :

1. **Applicability**. The specification technique should be applicable to a large set of different problems. Case studies of applying each technique to pratical problems are the main source to evaluate their applicability.

2. **Useability**. The specification language should be easy to learn and use. Support for multiple representations is one way to improve the useability. For example, graphical representation can aid in explaining the specifications.

3. The capability for specifying **nonfunctional requirements**, such as concurrency, security, reliability, performance, fault tolerance, and time-out.

4. **Verifiability**. The specification methodology should provide capability for validation of completeness, consistency, and correctness with respect to both syntax and semantics. The desired supporting tools includes : syntax checker, interpreter, theorem prover.

5. The capability for **equivalence checking**. The capability to study the equivalence between two independently created specifications will help check the consistency of the understanding of the informal requirements. This capability can also help prove the consistency between the specifications of different abstraction levels. The importance of equivalence notions in the context of formal descriptions of distributed systems has been widely recognized.

6. The support for **deriving implementation** from the specification, either automatically or through rigorous refinement steps.

In the assessment, Durra and Ina Jo + TL, instead of treating as new languages, are mentioned as the extension of Larch and FDM, respectively.

4.1 Applicability

VDM has been used for developing deterministic systems software, like compiler, database management systems, etc., as well as major parts of non-deterministic, concurrent and distributed software, such as operating systems, local area nets, office automation systems, etc. Many applications of

VDM can be found in [BJ87, B+88, JS90]. A number of projects in the application of Z, held at Oxford from 1978-1986, were reported in [Hay87]; the applications include the IBM's Customer Information Control System (CICS), UNIX filing system and distributed operating system. FDM has been applied in the formal specification of a number of large systems where the security properties were verified.

OBJ has been used for many applications, including debugging algebraic specifications [GCG90], rapid prototyping, specifying software systems (e.g, the GKS graphics kernel system, an Ada configuration manager, the MacIntosh QuickDraw program). Larch has similar expressive capability as OBJ, but few case studies using Larch were found in the literature. Based on the equational logic, both OBJ and Larch have been used for circuit verification [Gog88, G+88].

In SEDOS project, Estelle and LOTOS have been used to specify several ISO protocol and service standards [DV89]. SDL hse been used extensively for specifying telecommunication switching systems, and also found to be well-suited for real-time, distributed, and interactive systems. Temporal logic has been used for the specification and verification of concurrent program behavior [Lam83], reactive systems [Pnu86], real-time systems [Ost89] and hardware design [Mos85]. Few case studies using PAISLey, RTL, or SEGRAS are found in the literature.

4.2 Useability

Understandability appears to be inversely proportional to the level of complexity and formality present. In the study of [Dav88], Statecharts, PAISLey, and Petri nets appear to be much more difficult to comprehend than the others[3] which are mostly less formal. Roughly speaking, learning a definitional specification initially takes more practice than learning an operational specification, because programmers trained in conventional languages tend to think imperatively. However, it is very difficult to determine whether one technique has higher useability than the other since many factors have to consider, including human factors. In the following, we intend to evaluate useability of each technique based on (1) modularity and reusability of components, (2) support of "human-friendly" form, such as diagrams and flow charts, and (3) management tools for specification construction.

Z, OBJ and Larch support modularity and encourage reuse of components by providing libraries and mechanisms for parameterization, renaming, export-import interface, etc. BSI/VDM Specification Language, a language currently under standardization, will enchance original VDM for supporting modularity and parameterization [LA89]. In [Ber86], FDM was enchanced to support modularity. Most techniques using process paradigm or hybrid approach support modularity to some extent, but do not encourage reuse of components; RTL and temporal logic are two exceptions which do not support modularity. RTL describes timing properties of a system in a global way; temporal logic was traditionally used in a global, non-modular and non-compositional way since it reasons about the global state of the program, but some researchers have been investigating methods of the modular specification using temporal logic [HBP84, Lam83]. Modularity is only supported a little in PAISLey since process is the unique structuring unit. SDL and LOTOS support modularity and use algebraic technique, which encourage reusability, for specifying abstract data types. SEGRAS supports modularity and encourages reuse in both system structure (for dynamic behavior) and data structure.

Most surveyed specification techniques lack the support of "human-friendly" form, and only SDL

[3]The others include natural language, finite state machine, SA/RT, REVS, RLP and SDL.

and SEGRAS support graphical representation. A tool, called GROPE (Graphical Representation Of Protocols in Estelle), was prototyped with the intention to animate Estelle specifications in graphical form [NA90].

Syntax directed editors exist for some languages, such as Larch, Estelle, LOTOS, SDL and SEGRAS. In ESPRIT SEDOS project, the workstations of Estelle and LOTOS, for increasing efficiency and productivity in the development phases, have been prototyped [DV89]. SDL is the language in widest use in industry by specifiers and developers of telecommunication systems, such that diverse commercial supporting tools have been developed; YAST (Yet Another SDL Tool) [Z$^+$89], for example, is a set of tools that support the use of full SDL'88, including graphical editor, on-line SDL tutorial. The SEGRAS laboratory, an interactive specification environment, is designed to support the stepwise development of large specifications in SEGRAS [Kra87].

4.3 Nonfunctional Requirements

Most nonfunctional requirements, also called *constraints*, are difficult to be specified formally. Given the current state of the art, only some types of constraints are addressed by current formal specification techniques, such as security, concurrency, and timing constraints.

Some security properties can be verified with respect to functional specifications using the techniques with data paradigm, although they do not intend to provide constructs for specifying nonfunctional requirements. A large portion of the current formal verification work has been dominated by security related projects, and FDM is one of the techniques which have been used extensively in this area [C$^+$81]. In [WN89], Ina Jo was extended with temporal logic for specifying concurrency. Larch has been used to demonstrate the applicability of specifying some nonfunctional properties, such as synchronization [BHL87], persistence and atomicity [WG89]. In the work of Durra [BW87], Larch was extended with an event expression language for specifying timing behavior.

All the techniques using process paradigm or hybrid approach deal with concurrency. When describing timing behavior we usually want to be able to specify that (a) something should occur within a certain time otherwise ......, or that (b) after a certain time something must occur. In Estelle, a delay mechanism, which can specify the delay time for each enabled transition in a finite state machine, is provided for modeling time-out behavior (related to (a)) or waiting function (related to (b)). In SDL, timing behavior is described by setting a watch-dog timer which can be made in three different ways : (1) using a timing device, within or outside of the system that takes care of waking up the process at appropriate time, (2) using a continuous signal[4], (3) using the SDL construct of timer[5]. PAISLey provides only method (1) of SDL, which is the most cumbersome way, for modeling time-out behavior. Both PAISLey and SDL provide facilities for describing some kinds of performance requirements; PAISLey provides timing attribute attached to functions in the functional specification, referring to the evaluation time of the functions in a form of a random variable with lower/upper bounds, mean, or the distibution; different types of transmission delay can be modeled in SDL [SSR89]. Estelle, temporal logic, LOTOS and SEGRAS support the representation of time ordering aspects and handle concurrency well, but provide no construct for time measures; some real-time extensions of temporal logic [PH88, Ost89] and LOTOS [QF87] for expressing time quantitatively have been proposed. RTL is desgined specifically for specifying

[4]Built-in operator **NOW** can be used in any expression yielding the value of the current time.

[5]SDL timer is an entity which can be activated by the **SET** statement and produce a signal upon the expiring of the time set.

timing behavior and performance requirments.

4.4 Verifiability

Most techniques using data paradigm, based on either an extension of first-order predicate calculus or equational logic, provide a proof theory; all the theorem provers, primarily are proof-checker and bookkeeper, provide an environment for supporting formal reasoning where the human guides proof creation using their insight into the problem domain. OBJ is the only exception which provides interpreter instead of theorem prover; despite being incomplete, testing using interpreter is a practical way of increasing confidence in the correctness of the specification. In [Gog88], the technique for proving theorems using OBJ and its interpreter was developed.

Simulators for dynamic behavior checking exist for PAISLey, Estelle, LOTOS, SEGRAS[6] and SDL. Extensive efforts spent on the verification techniques for standard FDTs have produced some other verification tools for Estelle [D$^+$89], LOTOS [EVD89] and SDL [FM89]. Temporal logic provides sound *global* proof systems[7] for reasoning the properties of *entire* systems, but the technique to support compositional proof systems based on modular specifications is still under *intensive investigation* [HBP84]. In order to use temporal logic iteself as a tool for programming and simulation, two programming languages based on Interval Temporal Logic (ITL), named *Tempura* [Mos86] and *Tokio* [F$^+$86], were designed and implemented. A decision procedure for RTL formulas, although inherently computationally expensive, has been proposed in [JM86] for safety analysis.

4.5 Equivalence Checking

Among the surveyed techniques, only LOTOS, which is based on a process algebra derived from CCS, has developed the equivalence theories, using the notions of behavioral equivalence, and implemented the tool for equivalence checking[BC89]. In the work of [VB89], the mapping of SDL processes and queues onto an extended version of CCS was proposed to make equivalence checking possible[8]. The equivalence notion of algebraic specifications, which could be applied to OBJ and Larch, has been addressed in several studies [BW88].

4.6 Support for Deriving Implementation

Both VDM and FDM support stepwise refinement techniques for deriving implementation. Some theoretical studies have been done on the stepwise refinement methods for Z [Mor90] and OBJ [NF89]. The issue of translating OBJ notations into an efficient implementation is still a big challenge; in [Shu89], a development strategy was designed for translating OBJ into MALPAS intermediate language which is then refined until it is easily translated into code. Larch, based on a two-tiered specification technique, allows some implementation issues, such as modular decomposition and exception handling, to be specified in interface languages.

[6]An interactive Petri net simulator of SEGRAS was under developement as indicated in [Kra87].

[7]The proof systems are referred as *global* since they are only applicabl to entire systems, and cannot be applied to components of systems.

[8]CCS is a process algebra which has a clearly defined equivalence relation between processes. SDL, however, is a language and not an algebra, such that there is no way of telling whether two SDL specifications are equivalent (apart from their being extually identical).

The transformational methods for translating PAISLey specification into implementation, as proposed in [Zav84] where operational approach for software development was advocated, are not available yet. A generator of C source code for Estelle was built for both simulation and implementation purposes [RC89]. A set of tools, called *LIW* (LOTOS Implementation Workbench) [M+89], have been designed for providing an interactive process to refine a LOTOS specification into a C source code; the very high level abstraction of LOTOS precludes a direct compilation. For SDL, several implementation tools have been built [FM89], including CHILL source code generators, and code synthesizer generating C++ source code from SDL-PR. The development methodology for reactive systems based on temporal logic was discussed a little in [Pnu86], and the need of a compositional proof system as a prerequisite for such methodology was pointed out. The formal grounds of stepwise implemetation using SEGRAS were ongoing research.

5 CONCLUSION

Some active research issues on formal specification techniques are identified as follows : (1) hybrid specification methods, (2) model checking techniques, (3) object-oriented specification techniques. (4) specification languages for real-time systems, and (5) theory and practice on compositional methods and stepwise refinement techniques.

It is interesting that LOTOS, SDL and SEGRAS all adopt algebraic technique for describing properties of data domain at the lower tier, similar to Larch, and use other formalisms for specifying dynamic system behavior at the upper tier. This kind of two-tiered approach seems to become a general solution for combining the best world of data paradigm and process paradigm.

Model checking has become a well known method to carry out automatic verification of distributed systems. In this method, a model representing the behavior of the system is described using certain operational approach (serve as a behavioral specification), and the desired properties of the system are specified in temporal logic formulas (serve as a requirement specification). EMC [CES86] and **XESAR** [R+87] are two typical systems which use a subset of CSP and a variant of Estelle, respectively, for implementing the behavioral model, and use branching time temporal logic for specifying the desire proeprties. For verification, a complete state graph representing all the behaviors of the system is generated from the model first, and then a model checking algorithm is applied to check if the state graph satisfies the temporal logic formulas.

Most of these surveyed specification techniques have reached certain level of maturity, but many challenges still remain, such as building sophisticated tools, specifying real-time properties and nonfunctional requirements. We do not expect to develop a specification technique which is suitable for all classes of applications; instead, it is the job of specifiers to choose the appropriate technique given the problem at hand.

References

[Alf85] M. Alford. SREM at the age of eight: The distributed computing design system. *IEEE Computer*, 18(4):36–46, April 1985.

[B+88] R. Bloomfield et al., editors. *VDM'88: VDM - The Way Ahead*, volume 328 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988. VDM Europe Symposium 1988.

[BC89] T. Bolognesi and M. Caneve. Equivalence verification : Theory, algorithms and a tool. In P. H. J. van Eijk et al., editors, *The Formal Description Technique : LOTOS*, pages 303–326. North-Holland, 1989.

[Ber86] D. M. Berry. Adding modularity and separate subsystem specification support to the formal development methodology (FDM). Technical Report SP-4361, System Development Corp., Santa Monica, CA, March 1986.

[Ber87] D. M. Berry. Towards a formal basis for the formal development method and the Ina Jo specification language. *IEEE Transactions on Software Engineering*, SE-13(2):184–200, February 1987.

[BH89] F. Belina and D. Hogrefe. The CCITT-Specification and Description Language SDL. *Computer Networks and ISDN Systems*, 16(4):311–341, 1988/89.

[BHL87] A. Birrell, J. Horning, and R. Levin. Synchronization primitives for a multiprocessor: A formal specification. *Proc. of the Eleventh ACM Symposium on Operating Systems Principles*, pages 94–102, 1987. ACM/SIGOPS.

[BJ87] D. Bjørner and C. B. Jones, editors. *VDM'87: VDM - A Formal Method at Work*, volume 252 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987. VDM Europe Symposium 1987.

[BW87] M. R. Barbacci and J. M. Wing. Specifying functional and timing behavior for real-time applications. In *Proc. of the Parallel Architecture and Languages Europe, Vol II*, pages 124–140. Lecture Notes in Computer Science, vol. 259, 1987.

[BW88] F. L. Bauer and M. Wirsing. Crypt-equivalent algebraic specifications. *Acta Informatica*, 25(2):111–153, February 1988.

[C$^+$81] M. H. Cheheyl et al. Verifying security. *ACM Computing Surveys*, 13(3):279–339, September 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[D$^+$89] M. Diaz et al. *The Formal Description Technique Estelle*. North-Holland, 1989.

[Dav88] A. M. Davis. A comparison of techniques for the specification of external system behavior. *Communications ACM*, 31(9):1098–1115, September 1988.

[DeM78] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, 1978.

[DV89] M. Diaz and C. Vissers. SEDOS : Designing open distributed systems. *IEEE Software*, 6(5):24–33, November 1989.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification*. Springer-Verlag, 1985.

[EVD89] P. V. Eijk, C. A. Vissers, and M. Diaz. *The Formal Description Technique LOTOS*. North Holland, 1989.

[F$^+$86] M. Fujita et al. Tokio : Logic programming language based on temporal logic and its complilation to prolog. In E. Shapiro, editor, *Proc. 3rd International Conference on Logic Programming*, pages 695–709. Lecture Notes in Computer Science, vol. 225, Springer-Verlag, 1986.

[FM89] Ove Færgemand and M. M. Marques, editors. *SDL '89 : The Language at Work*. North-Holland, 1989.

[G$^+$88] S. J. Garland et al. Verification of VLSI circuits using LP. In *Proc. IFIP WG 10.2, The Fusion of Hardware Design and Verification*. North Holland, 1988.

[Gal87] A. Galton, editor. *Temporal Logics and their Applications*. Academic Press, 1987.

[GCG90] C. P. Gerrard, D. Coleman, and R. M. Gallimore. Formal specification and design time testing. *IEEE Transactions on Software Engineering*, 16(1):1–12, January 1990.

[GHW85] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report 5, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, July 1985.

[Gog84] J. A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, September 1984.

[Gog88] J. A. Goguen. OBJ as theorem prover with applications to hardware verification. Technical Report SRI-CSL-88-4R2, SRI Computer Science Lab, August 1988.

[GW88] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI Computer Science Lab, August 1988.

[Hal86] B. Halipern, editor. *Special Issue on Multiparadigm Languages and Environments*. IEEE Software, January 1986.

[Har87] D. Harel. Statecharts : a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Hay87] I. Hayes, editor. *Specification Case Studies*. Prentice Hall, Inc., 1987.

[HBP84] R. Kuiper H. Barringer and A. Pnueli. Now you may compose temporal logic specifications. In *Proc. 16th ACM Symp. on Theory of Computing*, pages 51–63, 1984.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Inc., 1985.

[Jac88] D. Jackson. Composing data and process descriptions in the design of software systems. Technical Report MIT/LCS/TR-419, Department of Computer Science, MIT, May 1988.

[JM86] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[Jon89] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, Inc., second edition, 1989.

[JS90] C. B. Jones and R. C. Shaw, editors. *Case Studies in Systematic Software Development*. Prentice Hall, Inc., 1990.

[Kra87] B. Kramer. SEGRAS - a formal and semigraphical language combining petri nets and abstract data types for the specification of distributed systems. In *Proc. 9th International Conference on Software Engineering*, pages 116–125, 1987.

[LA89] P. G. Larsen and M. M. Arentoft. Towards a formal semantics of the BSI/VDM specification language. In *Information Processing 89*, pages 95–100. IFIP, 1989.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[M+89] José A. Mañas et al. The implementation of a specification language for OSI systems. In P. H. J. van Eijk et al., editors, *The Formal Description Technique : LOTOS*, pages 409–421. North-Holland, 1989.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Mor90] C. Morgan. *Programming from Specifications.* Prentice Hall, Inc., 1990.

[Mos85] B. C. Moszkowski. A temporal logic for multi-level reasoning about hardware. *IEEE Computer*, 18(2):10–19, February 1985.

[Mos86] B. C. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, 1986.

[NA90] D. New and P. D. Amer. Adding graphics and animation to Estelle. *Information and Software Technology*, 32(2):149–161, April 1990.

[NF89] A. T. Nakagawa and K. Futatsugi. Stepwise refinement process with modularity : an algebraic approach. In *Proc. 11th International Conference on Software Engineering*, pages 166–177, 1989.

[Ost89] J. S. Ostroff. *Temporal Logic for Real-Time Systems.* Research Studies Press LTD., 1989.

[PH88] A. Pnueli and E. Harel. Applications of temporal logic to the specification of real-time systems. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 84–98. Lecture Notes in Computer Science, vol. 331, Springer-Verlag, 1988.

[Pnu86] A. Pnueli. Specification and development of reactive systems. In H. J. Kugler, editor, *Information Processing 86*, pages 845–858, 1986.

[QF87] J. Quemada and A. Fernandez. Introduction of quantitative relative time into LOTOS. In H. Rudin and C. H. West, editors, *Proc. 7th IFIP Conference on Protocol Specification, Testing and Verification*, pages 105–121, 1987.

[R+87] J. Richier et al. Verification in XESAR of the sliding window protocol. In H. Rudin and C. H. West, editors, *Proc. 7th IFIP Conference on Protocol Specification, Testing and Verification*, pages 235–248, 1987.

[RC89] Jean-Luc Richard and T. Claes. A generator of c-code for Estelle. In M. Diaz et al., editors, *The Formal Description Technique : Estelle*, pages 397–420. North-Holland, 1989.

[Shu89] R. N. Shutt. A rigorous development strategy using the OBJ specification language and the MALPAS program analysis tool. In C. Ghezzi and J. A. McDermid, editors, *ESEC '89, 2nd European Software Engineering Conference*, pages 260–291. Lecture Notes in Computer Science, vol. 387, Springer-Verlag, 1989.

[SSR89] R. Saracco, J. R. W. Smith, and R. Reed. *Telecommunications Systems Engineering using SDL.* North-Holland, 1989.

[TH77] D. Teichroew and E. A. Hershey. PSL/PSA: A computer aided technique for structured documentation and analysis of information processing systems. *IEEE Transactions on Software Engineering*, 3(1):41–48, January 1977.

[VB89] Pieter H. A. Venemans and Rob A. Beukers. An experiment with algebraic reduction. In Ove Færgemand and M. M. Marques, editors, *SDL '89 : The Language at Work*. pages 325–334. North-Holland, 1989.

[WG89] J. M. Wing and C. Gong. Machine-assisted proofs of properties of Avalon programs. Technical Report CMU-CS-89-172, Department of Computer Science, Carnegie Mellon University, August 1989.

[WN89] J. M. Wing and M. R. Nixon. Extending Ina Jo with temporal logic. *IEEE Transactions on Software Engineering*, SE-15(2):181–197, February 1989.

[Z$^+$89] M. Zorič et al. Tool set development and the use of SDL. In Ove Færgemand and M. M. Marques, editors, *SDL '89 : The Language at Work*, pages 77–86. North-Holland, 1989.

[Zav84] P. Zave. The operational versus the conventional approach to software development. *Communications ACM*, 27(2):104–118, February 1984.

[Zav89] P. Zave. A compositional approach to multiparadigm programming. *IEEE Software*. 6(5):15–25, September 1989.

[ZS86] P. Zave and W. Schell. Salient features of an executable specification language and its environment. *IEEE Transactions on Software Engineering*, SE-12(2):312–325, February 1986.

SUPPORT FOR SOFTWARE DESIGN, DEVELOPMENT, AND REUSE THROUGH AN EXAMPLE-BASED ENVIRONMENT

Lisa Neal†
EDS Center for Machine Intelligence
One Cambridge Center
Suite 408
Cambridge, MA 02142
email: lisa@cmi.com
tel: (617) 225-0095

ABSTRACT

Example-based programming is a methodology which supports software design, development, and reuse. The integration of examples into a software development environment results in a tool which is especially effective for the support of experienced software developers who are working in new domains. Example programs are used as instances of language constructs, thus providing syntactic information through instantiations of templates, or as examples of algorithms or programs. Examples selected from a library can be viewed, totally or partially copied, or run. The initial example-based programming environment was implemented for Pascal on the Macintosh computer. The system was successful in addressing problems with the use of structure editors and in facilitating the reuse of software. In its initial conception, example-based programming was designed to provide passive assistance in the software development process; however, it is a paradigm which is extensible to provide more active and intelligent support. The initial system and results from an empirical study are presented, and current and future developments are discussed, with a focus on extensions which increase the amount and types of support provided by the environment.

† This research was supported, in part, by U.S. Army Research Office Grant #DAAG29-83-G0008 and, in part, by the Defense Advanced Projects Research Agency under contract N00014-85-C-0710. I would especially like to thank Tom Cheatham and Ugo Gagliardi for their support of this work.

1. INTRODUCTION

We define an example-based environment as one which incorporates examples to facilitate the design, development, and reuse of software. The concept of integrating examples into an environment arose from empirical studies of structure editors [Neal 87a, 87b]. The editors under study, which were oriented primarily toward novice programmers, were designed to support the construction of Pascal programs through a structured and restrictive approach to input, modification, and navigation. Templates were available for all language constructs, and were the means by which programs were entered.

The highly structured approach to program construction and editing was limited in its success because users lack thorough knowledge of the structure of their program and its underlying representation. More experienced users, in particular, found the restrictiveness of the editors frustrating, especially when they were familiar with and comfortable with the use of non-language-based text editors [Neal and Szwillus 90]. The template approach to input theoretically had advantages over textual input due to factors such as less focus on syntactic details and a greater emphasis on semantics and algorithms. In fact, template insertion was often cumbersome or awkward, and, more importantly, did not effectively prompt users.

In examining alternate approaches to structure editing which both aid users more effectively and take a less restrictive approach, it was considered likely that program fragments, as, in essence, instantiations of templates, would prompt users more effectively than templates. However, program fragments lack the contextual information, such as declarations, necessary to understand and effectively utilize a program fragment. Hence, it was determined that actual examples of programs were likely to provide more useful information. The use of preexisting code is common in many instances: novices use code in textbooks and their previously written programs, experienced programmers use their own and other's code, and teams of programmers share code. However, access to preexisting code is not actively supported by most programming environments.

While it seemed likely that this approach would be beneficial for development, the benefits for design and reuse were only discovered following the implementation of the example-based environment. Likewise, empirical evidence showed that this approach was especially beneficial for programmers who are experienced but are working in a new domain. Since novices do not remain novices for very long, but experts are always becoming novices in a new domain or have domains in which they work infrequently, we felt it was important to provide support within a programming environment for this population [Neal and Szwillus 90].

2. INITIAL IMPLEMENTATION

A syntax-directed editor for Pascal on the Macintosh was augmented with an example window and access to an example library. The editor uses a palette to present the available templates and has a command completion capability which allows the enter key to be used after typing a keyword in order to invoke a template. In addition, textual input is incrementally parsed upon the use of a semi-colon or the enter key. The editor hence provides templates for input, but allows free textual input as well. Modification and navigation can likewise be either structural or textual.

Examples in a separate example window can be viewed, totally or partially copied, or run. The only restriction on use of the example window was that edited examples could not be saved to the same program name, in order to preserve the quality of the library. For the initial implementation, examples were accessed only through descriptive program names. Examples are selected through a dialog window which has unlimited size.

3. EMPIRICAL STUDY

An empirical study was performed with the initial version of the system in order to determine how the examples were used and their effectiveness [Neal 89]. It became clear through the study that the example-based system had the potential to aid in comprehension, design, and reuse as well as program construction.

The primary results of the study follow. The examples were more heavily used than any of the structure-based capabilities of the syntax-directed editor and provided capabilities lacking in the syntax-directed editor. The examples were used in a variety of ways: to aid in design, to aid in writing code, to aid in comprehension of syntax and semantics, and for reuse. Specifically, examples were used: to determine the use of Macintosh-specific language features and procedures; for I/O, both syntactically and semantically; to discover approaches to solving problems; to be executed in order to better understand or to test an understanding of the semantics of a program; for inspiration and guidance towards a solution; and as guidelines for formatting and standardizing code.

While the participants in the empirical study were given a constrained task rather than given the environment to use in solving a problem on which they were already working, the results were strong enough and the feedback enthusiastic enough that we were encouraged to pursue the approach further based on a generalization of the results. We were especially encouraged by the response of experienced programmers who were novice or infrequent users of Pascal. They found that they were easily able to construct Pascal programs which they understood and had confidence in the correctness of, rather than the process being a struggle which resulted in poorly written and semantically incorrect code.

4. USE OF AN EXAMPLE-BASED SYSTEM

From the empirical study and from observation of more casual use of the system, we found that the example-based approach is effective in a number of ways. Examples facilitate the learning process [Lieberman 86] and are essential to processes such as case-based reasoning, in which known solutions are adapted to solve problems. It is very common for textbooks and manuals to make heavy use of examples and for system developers to use examples from these sources, as well as their own or other's code. When an experienced programmer is a new or infrequent user of a programming language, examples are used to learn or refresh knowledge of a language. Likewise, examples are effective for starting in new domains, such as the use of windowing systems or interface builders, or for programming on a parallel architecture on which the programming languages may be new or hybrid languages. Once a programmer has developed expertise in one or more languages, learning another language by looking at existing programs and extracting the relevant information is not especially difficult and is less time-consuming than using a tutorial or reading a textbook, which is likely to give too much information at a greater level of detail than is needed.

The example-based environment has been successful in helping users in the design process. The availability of examples within the editing environment means that a user can easily scan examples, studying approaches to solving similar problems. Examples in the example library are well-written and well-documented, aiding in this process. Examples can be reused in part or in full, through the cut-and-paste facility or through procedure and function calls. Reuse is encouraged because of the accessibility of examples and because examples can be viewed and run, which allows a programmer to feel that the routine is much better understood than if the only access is to a routine name and parameter list. Additional support for design and reuse is included in enhancements to the system.

5. CURRENT AND FUTURE DIRECTIONS

Current development of example-based programming system is in two directions: new domains and system enhancements. The former includes support for programming on the Connection Machine for non-parallel programmers and support for E-L. The latter includes additional capabilities and support; for instance, the system is being enhanced to allow multiple access mechanisms for the example library and to annotate examples with information related to the design and use of the artifact.

The initial mechanism for access to examples, through descriptive program names, proved to be effective but too simplistic. When it became clear through observations of use that users often wanted an example of a particular language construct without having to scan multiple examples, access to examples through a hypertext-like capability was added, where a language construct could be selected in the main editing window and an example would be displayed in the example window which included the use of the construct. Examples were linked so that a number of examples would be made available, with simple examples first, followed by more complex instances of the construct. The linking of the examples had to be done explicitly as additions were made to the example library.

In considering more sophisticated access mechanisms, we rejected approaches which overly constrained the user in the process of coding or in the use of the system. We chose the use of an embedded design language, which is natural language in comments within the code being written. Guidance is provided through a help window for the optimal use of comments, and keywords are suggested. Also, examples provide additional guidance. For example, a program to sort would include in its comment the word 'sort.' In the same manner that selection of syntactic elements invoke examples including those elements, the use of 'sort' would invoke examples including that within their comments. This approach has the advantages of supporting the inclusion of design information without forcing the user to code in a particular way or use an artificial language, and of extending the retrieval capabilities of the system. The better access to examples the user has, the more likely it is that they will be taken advantage of as code to be examined or reused.

Additional enhancements include annotations to examples. Examples can become large, and the annotations provide additional, selectable information without increasing the example size. Annotations include design information, explanations of syntax, snapshots of memory, and instances of inputs and outputs. While the embedded design language can be used to provide well-documented routines, we wanted to include more extensive information which recoded the design history and alternatives [MacLean, Bellotti, and Young 90]. This provides information which allows code to be more effectively understood and reused, and is especially desirable for the maintenance and enhancement of code. In addition, we are exploring more graphical approaches for specifying design [Szwillus 89]. The other annotations are primarily to allow easy access to information which may prove useful and can increase a user's understanding of a routine or its components with as little effort as possible. For instance, even though examples can be run, the availability of sample inputs and outputs means that the user need not switch contexts in order to find out more about what a program does or what its output looks like.

Future directions for the example-based environment include further work on system enhancements. The inclusion of a more knowledge-based approach to the use of examples is being explored. We would like to incorporate access to relevant examples through an intelligent assistant that would monitor the user's input and example use in order to determine what the user's goals and plans are and provide appropriate examples to aid the user in achieving that goal. Such approaches have been successful in systems dealing exclusively with semantic knowledge [Johnson and Soloway 85]. More semantics-based retrieval could aid in one of the significant problems with software reuse: the identification of appropriate and relevant reusable elements [Tracz 87]. Attempts to automate assistance have met with limited success, hence we are examining approaches in which the user is in control but with intelligent support.

While the syntax-directed editor integrated into the example-based environment has not been especially successful in aiding in design or development, we would like to further consider how to effectively utilize the knowledge about a language that structure editors have. One approach is to think of a structure editor as a knowledgeable assistant, which monitors a user's actions, and guides, rather than forces, a user. The structure editor could give experts advice when requested and give novices continual feedback, acting more like an unobtrusive intelligent tutoring system for designing and programming. Alternately, the structure editor can be used to aid more in understanding code than in writing code; views of the code which reveal its structure or provide condensed views can aid in a higher level understanding.

6. CONCLUSIONS

Example-based programming was originally conceived as a supplement to structure editors. Empirical evidence showed the effectiveness of the example-based approach for the design, construction, and reuse of code, and its superiority over some of the capabilities of structure editors. Current and future enhancements to the system are increasing the amount of information included in examples and the accessibility of relevant examples. The example-based approach is showing effectiveness particularly for experienced programmers working in new or infrequently used domains, since the information provided by the examples allow a programmer to more easily get started through the extraction of useful and relevant information. The examples provide support for the design process and encourage the reuse of existing code, in addition to supporting the development of code.

7. REFERENCES

Johnson, W. L. and Soloway, E. PROUST: Knowledge-based program understanding, *IEEE Transactions on Software Engineering* Volume SE-11, March 1985.

Lieberman, H. An Example Based Environment for Beginning Programmers. *Instructional Science* 14 (1986).

MacLean, A, Bellotti, V, and Young, R. What Rationale is there in Design? To appear in *Proceedings of INTERACT'90 3rd IFIP Conference on Human-Computer Interaction* (Cambridge, England, August 27-31, 1990).

Neal, L. R. Cognition-Sensitive Design and User Modeling for Syntax-Directed Editors. In *Proceedings of CHI+GI'87 Conference on Human Factors in Computing Systems and Graphics Interface* (Toronto, April 5-9). ACM, New York, 1987a.

Neal, L. R. User Modeling for Syntax-Directed Editors. In Bullinger, H.-J. and Shackel, B. (editors), *Human-Computer Interaction - INTERACT'87*. Amsterdam: North-Holland, 1987b.

Neal, L. R. A System for Example-Based Programming. In *Proceedings of CHI'89 Conference on Human Factors in Computing Systems* (Austin, April 30 - May 4). ACM, New York, 1989.

Neal, L. R. and Szwillus, G. Report on the CHI'90 Workshop on Structure Editors. To appear in *SIGCHI Bulletin,* Vol. 22, No. 2, October 1990.

Szwillus, G. Editing Graphical Structures. In Smith, M. J. and Salvendy, G. (Editors) *Work with Computers: Organizational, Management, Stress and Health Aspects.* Elsevier Science Publishers B.V., Amsterdam, 1989.

Tracz, W. Software Reuse: Motivators and Inhibitors. In *Proceedings of Compcon '87 (Spring), Thirty-second IEEE Computer Society International Conference,* February 23-27, 1987, Cathedral Hill Hotel, San Francisco, CA.

INTEGRATING CODE KNOWLEDGE WITH A SOFTWARE INFORMATION SYSTEM*

Peter G. Selfridge
AT&T Bell Laboratories
Murray Hill, NJ 07974
phone: 201-582-6801
e-mail: pgs@allegra.att.com

ABSTRACT

This paper describes CODE-BASE, a Software Information System designed to facilitate interactive discovery of information about an unfamiliar large body of software. CODE-BASE represents generic information about UNIX and C code and has access to a database of code objects and their relationships for a specific software project. By querying CODE-BASE a user can get quick access to various kinds of code information. More interestingly, CODE-BASE supports the creation of new code concepts which become part of the knowledge base and can be used in later queries.

1. Introduction

Large software systems are becoming increasingly important in modern life, controlling not just computers but defense systems, banks, telecommunications systems, and databases. At the same time, the problems in creating, understanding, maintaining, and extending such systems is becoming increasingly problematic. One common aspect of maintenance and other problems with large software is the **discovery** problem, which is the problem of learning enough about an existing system in order to use or modify it. Before attempting a particular task, a developer must often spend a great deal of time "discovering" features of the system, including the overall organization of the software and the conceptual framework that drove that organization and the location and details of specific functions and data structures.

The discovery process is clearly knowledge-intensive. Knowledge-based Software Information Systems (SIS's) can help discovery by encoding a variety of knowledge about an existing system and providing powerful ways of accessing that knowledge [3]. If built with appropriate knowledge representation technology, these systems can help by:

- organizing the underlying knowledge into a taxonomy of related concepts;
- automatically classifying new information into the existing taxonomy;
- allowing for inheritance of properties within the taxonomy;
- and,by classifying queries and computing the matching individuals, support "query by reformulation" [19], an interaction technique particularly well suited to the discovery process.

Previous experience with the LaSSIE system [9,10] emphasized the need for comprehensive links to the code itself (LaSSIE's primary emphasis was on a higher-level "action and object" domain model of a telecommunications switch). In fact, much of the discovery process seems to involve forming meaningful associations between parts of a high-level domain model and the code pieces that implement it. Discovery is an incremental process where the user asks questions that attempt to bridge this gab between the code objects, generated syntactically, and elements of a domain model. In doing so, the user may also expand or flesh out parts of this domain model in his or her own mind. For example, consider a body of code that controls a telecommunications switch and implements a number of interaction features. Imagine a software developer with the task of fixing a bug in the last-number-dialed (LND) feature. Such a developer might start off with the following general questions:

1. what function implement the LND feature?
2. what macros are involved in the feature and where are they defined?
3. where are errors handled?
4. what messages are invoked by this feature?

In order to answer questions of this sort, the user needs to be able to examine the source files, the code objects, like functions and macros, their call relationships, as well as information describing how the feature works at the user level. In the process of answering the above questions, it would be convenient to be able to create new concepts or categories so they could be used later. For example, the user might create the category **error function,** a subset of **function**, after discovering that all error functions reside in a particular directory. Next, the user might create the category **LND-function** after discovering that three particular functions implement the LND feature. The user can now ask what error-functions are called by any LND-function to discovery information relevant to question three above.

This paper describes CODE-BASE, a SIS designed to support the discovery process by initially representing automatically acquired code information and providing the kind of concept extension ability illustrated above. After a general description of the goals of this project and the technology used to implement it, we describe the kinds of code knowledge represented and how the knowledge is acquired, represented, stored, and accessed. Then, we elaborate a little more on how CODE-BASE is used for discovery with a sample scenario and describe how automatic classification facilitates this process. Finally, we briefly compare this project with other applications of knowledge representation technology to software problems and describe future work.

2. The CODE-BASE Project

The goals of the CODE-BASE project are to provide a knowledge-based Software Information System to facilitate the discovery process over large bodies of software. In order to do this, CODE-BASE represents in the Classic Knowledge Representation (KR) language a description of C code and generic UNIX information, as well as a small amount of project-specific knowledge. The user queries CODE-BASE in a query language and receives back from CODE-BASE a list of matching instances. The user can create new concepts or categories and populate them from the results of a query, as well as create combinations of old concepts. These new concepts can then be used in subsequent queries. The CODE-BASE graphical interface facilitates browsing of the knowledge base and several forms of hypertext-style interaction.

CODE-BASE is written on top of the Classic KR system [5, 6]. Classic provides a language for describing complex categories, relationships, and objects using three notions. First is the notion of a concept, which describes a class of objects in the world. Concepts are stored in a taxonomy which represents an is-a hierarchy and provides for the object-oriented inheritance of concept properties. The second is that of individual, which is an instance of a concept. Third, a role (or slot) captures various constraints between concepts and individuals. Roles can have associated with them constraints on their fillers of various kinds. For example, CODE-BASE contains the concept **c-file** which is a sub-concept of **code-object**, and includes a role called **has-directory** (since every UNIX file is contained in a directory) whose filler is constrained to be a **directory** (which is a sub-concept of **c-file**). Instances of **c-file** are particular c-files with properties derived from the definition of **c-file**.

Classic provides several limited kinds of active inference of the knowledge base. Most important are two kinds of automatic classification. Classification of concepts takes a new concept description and automatically places it in the proper part of the taxonomy. Classification of individuals is similar: given a new individual, Classic will determine which concepts that individual belongs to. Classic also provides inheritance of properties, contradiction detection (integrity checking), and simple forward chaining triggers.

Because of the amount of code knowledge that can be generated from even a modest number of source files, CODE-BASE is designed to run with all of the code knowledge stored initially in a database. We have implemented a scheme of **demand loading** of the knowledge in response to specific queries. In this scheme, the in-core system, which initially contains only the knowledge representation "schema", communicates with a database server to fetch specific objects and specific relationships from disk when they are needed. In addition, it is sometimes necessary to go back to the original source files to extract information not originally acquired; the database server handles this as well.

The overall architecture of CODE-BASE is shown in figure 1. The CODE-BASE front end (FE) communicates with the CODE-BASE Database Server (DS) through a set of UNIX pipes; these components are both loaded into core memory. When a user queries the system, the FE in turn queries the DS for the set of objects and relations necessary to respond to the query. The DS, in turn, loads information from one of the disk components (the code database or the source files), translates the information into Classic expressions, and returns them to the FE where they are evaluated. If some or all of the information has already been sent to the FE from prior requests, which the DS keeps track of, the information is not reloaded.

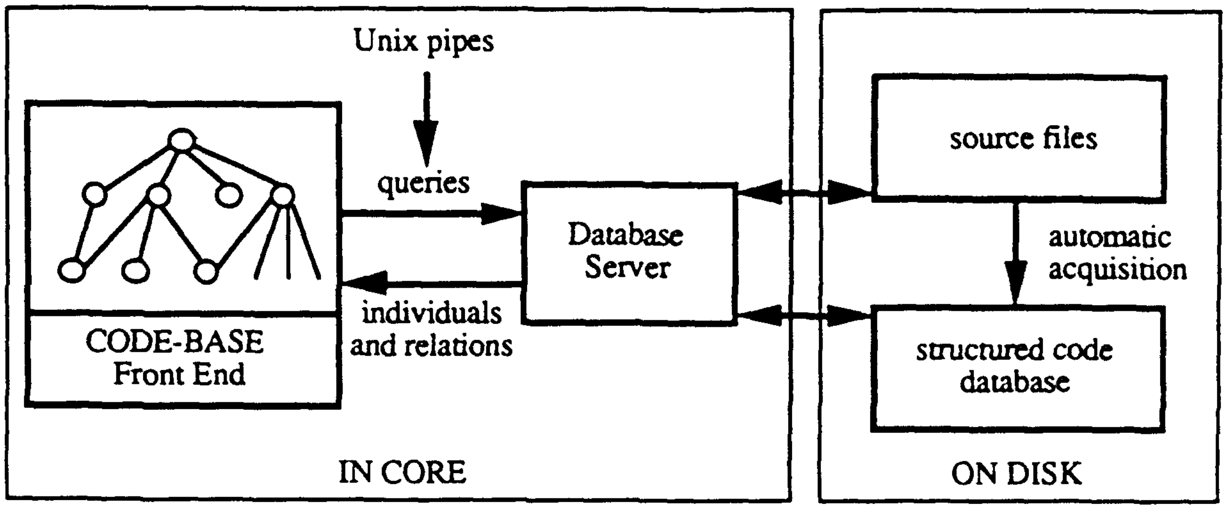


Figure 1: architecture of CODE-BASE

The remainder of this section describes:

- the code knowledge being stored;
- how the knowledge is acquired;
- how the knowledge is represented;
- how the knowledge is stored; and
- how the knowledge is accessed.

2.1 Code Knowledge

The software base we are representing is the Call Processing subsystem for the Definity 75/85 PBX - a mid-sized telecommunications switch [1]. This switch provides for "plain old telephone service" (POTS), dozens of special features like call-forwarding, conference calling, and the like, and the ability to customize some of the routing decisions for outside calls. The Call Processing subsystem (one of three – the other two being Maintenance and Administration) controls these functions and features for the users of the switch. It consists of several hundred thousand lines of C code in several thousand source files. The source files are compiled to produce about a dozen separate processes which communicate with messages to request services, communicate state, and coordinate process activity in the running switch.

There are three categories of code knowledge that are represented in CODE-BASE. The first is the file and directory structure of the Call Processing subsystem. The directory structure is fairly well-organized: sub-directories hold files that are compiled into the individual processes or shared libraries, and files related to a given feature (like call forwarding) or sets of features (like the "motel" package, which allows a user to request maid services, wakeup calls, et cetera) are usually grouped into directories. The second category of code knowledge is the definition and use of code objects, including files, functions, macros, type declarations, and global variables. This category of code knowledge is very large: a single process in the subsystem can have thousands of code objects and tens of thousands of relationships between these objects. Third, CODE-BASE represents the set of processes that make up the Call Processing subsystem and the set of messages that are sent between these processes.

2.2 Knowledge Acquisition

Automatic acquisition of code knowledge is critical to this enterprise: hand-engineering of most of the knowledge is not feasible because there is too much of it. Fortunately, most of the knowledge we are interested in is syntactic in nature and can be extracted from the set of source files with either general-purpose code analyzers or custom programs which are not difficult to write. In the first category is the CIA (C Information Abstraction) System [7], which extracts from a set of source files a set of code objects, where each object is defined (define relations), and every place where each object is used (reference relations). The output of CIA is a small set of (very large) disk files which are relational in structure; these make up the structured code database shown in figure 1. In the second category, we have written a program to traverse a hierarchical directory structure and produce a suitable encoding of that structure for loading into our Knowledge Representation (see below), as well as routines which augment the output of CIA in several ways.

At present, we have hand-coded the process structure of the Call Processing subsystem, which means entering into a disk file the names of the 13 separate processes and some ancillary information. The message information is extracted automatically from test scripts of messages taken off of a running switch; these scripts encode in ascii the sending process, receiving process, and message type. We do not currently represent the script as a sequence of messages, as is done in [16].

2.4 Knowledge Storage

All code knowledge generated by the CIA system is kept in the structured code database shown in figure 1. When the user queries the system (see the next section), Classic determines what objects and relations are needed to answer the query. If a set of objects are needed, the Classic side sends a simple request to the Database Server (DS), which in turn locates the objects in the structured code database, translates the information into Classic expressions, and sends them back to be evaluated. For a given individual, the role fillers are not loaded unless they are needed for a given query, although the values in all role descriptors are loaded. Essentially, we have implemented a set of "methods" which, if run, will call the DS to load all the slot fillers for that object. This is similar to other schemes, including "active values" in Common Lisp's CLOS package, and "triggers" in [18]. The DS keeps track of what objects and relations have already been loaded so they are not loaded again. The next section illustrates in a little more detail how this scheme works.

2.3 Knowledge Representation

The representation of code knowledge is done in the Classic Knowledge Representation System [5,6]. As described, Classic provides structured descriptions, called concepts, that can be organized in an "is-a" hierarchy, with inheritance of properties from less specific to more specific concepts. Each concept can have slots, called roles, which represent both intrinsic attributes of individuals of that concept and relationships between individuals; role fillers can be constrained in various ways as a type-checking and classification mechanism. Classic automatically classifies new concepts and individuals. As an example of classification, imagine the user creates the concept of "high-fanout function" and defines it as a function which calls more than 10 other functions. Classic will classify this new concept under the existing concept "function" and automatically classify all matching function individuals under this concept. Figure 2 shows a portion of the concept hierarchy (with no interrelations shown):

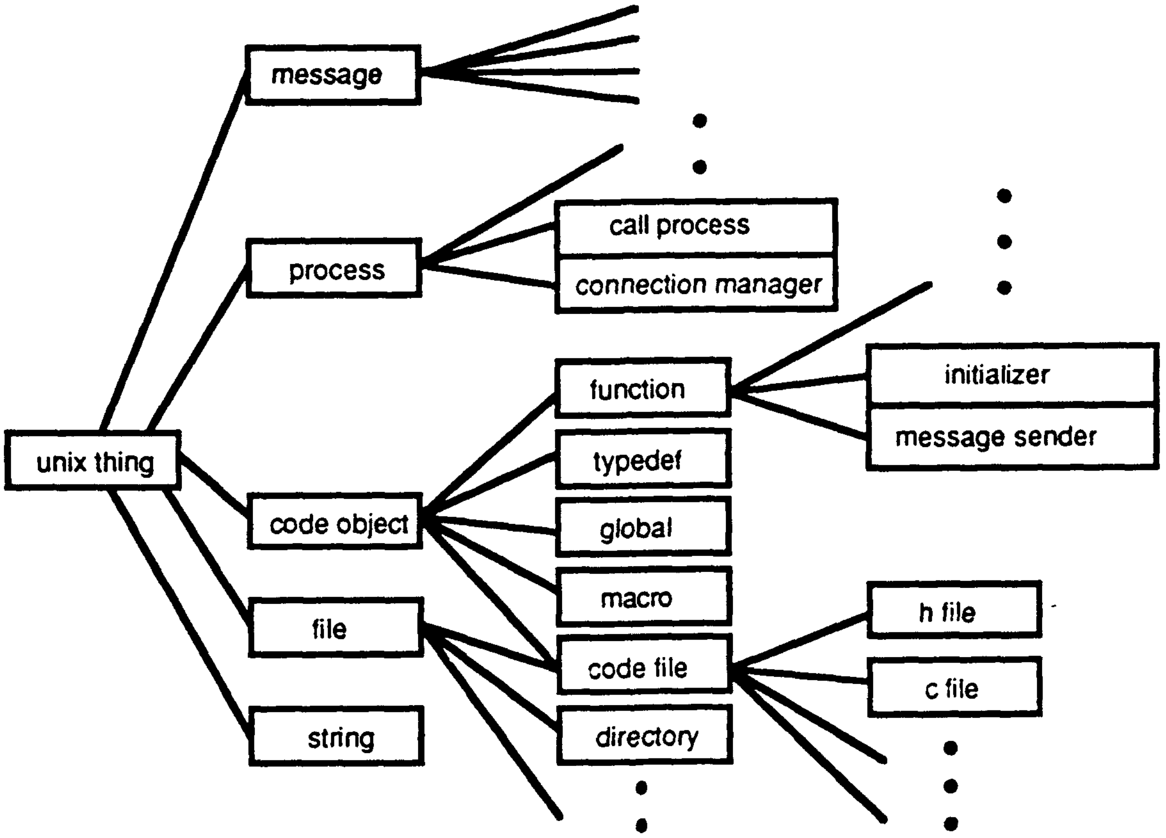


Figure 2: part of the CODE-BASE object hierarchy

Relations between individuals are represented in roles and role fillers. For example, the concept of file includes the role **has-directory**, which is restricted to contain a single value which must be a **directory**. This encodes the fact that in the Unix file system, all files are contained in a directory. For another example, all code objects have a role **has-defined-in**, which must be a **code-file**, encoding the fact that all code-objects (as extracted by CIA) must be defined in some code file. This role is further constrained if the code-object is a **macro**, in which case the code-file that defines it must be an **h-file**.

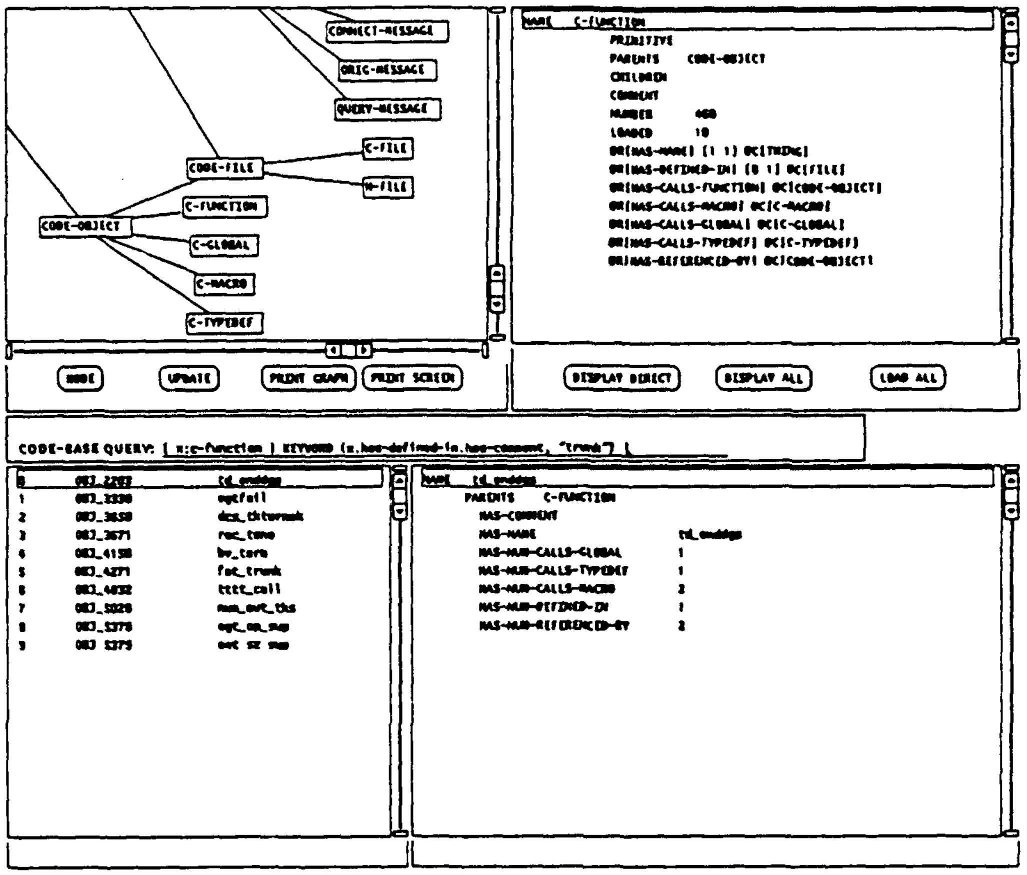
For a final example, all code-objects have the roles **has-references** and **has-referenced-by**, which contain other code-objects which reference and are referenced by that object, respectively. For functions, the **has-references** slot is further broken down into **has-references-function**, **has-references-macro**, **has-references-global**, and **has-references-typedef**.

One feature of Classic which impacts our database scheme, described in the next sub-section, is the ease in which it deals with partial descriptions. A partial description is an individual about which not all is known; Classic treats these just like other individuals, classifying them on the basis of what it knows at the time. When more information becomes available, Classic may re-classify them if indicated by the new information.

Keeping knowledge in both in core and on disk requires that CODE-BASE have a representation of what knowledge is available off-line. There are two kinds of knowledge in this category: the set of individuals which are instances of a particular concept; and the set of individuals which fill a particular role of an individual. For each of these kinds of knowledge, CODE-BASE has "meta-knowledge" about the number of individuals or role fillers that exist. For knowledge about which individuals are instances of a particular concept, each concept in the hierarchy has an associated "meta-concept" that represents the number of individuals that are instances of that concept. This is possible because we have a closed world, and the number of instances of each concept can be pre-computed and loaded with the schema. (Note: these "meta-concepts" are not shown in figure 2.) Knowledge about role fillers, is handled in a similar fashion. Each role in every individual has an associated attribute (a role with only one value) called a role descriptor that holds the number of role fillers that exist. Again, this can be pre-computed and becomes another piece of information about that individual. In fact, this kind of information can be useful in this domain – it may be useful to know that a given function calls 126 (or 0) other functions without needing to know anything else about those called functions (except for their existence).

2.5 Knowledge Access

CODE-BASE is embedded in a graphical interface with 5 components, shown in figure 3:



```
NAME    C-FUNCTION
        PRIMITIVE
        PARENTS    CODE-OBJECT
        CHILDREN
        COMMENT
        NUMBER     400
        LOADED     10
        OR[HAS-NAME] [1 1] OC[THING]
        OR[HAS-DEFINED-IN] [0 1] OC[FILE]
        OR[HAS-CALLS-FUNCTION] OC[CODE-OBJECT]
        OR[HAS-CALLS-MACRO] OC[C-MACRO]
        OR[HAS-CALLS-GLOBAL] OC[C-GLOBAL]
        OR[HAS-CALLS-TYPEDEF] OC[C-TYPEDEF]
        OR[HAS-REFERENCED-BY] OC[CODE-OBJECT]
```

```
CODE-BASE QUERY: [ x:c-function ] KEYWORD (x.has-defined-in.has-comment, "trunk") [
```

```
0    OBJ_2203    td_onddgs
1    OBJ_3330    eqtfail
2    OBJ_3658    dcs_tktarnak
3    OBJ_3671    rec_tune
4    OBJ_4158    bv_tare
5    OBJ_4271    fat_trunk
6    OBJ_4832    tttt_call
7    OBJ_5028    mam_ovt_ths
8    OBJ_5378    eqt_on_map
9    OBJ_5375    ext_sz_map
```

```
NAME    td_onddgs
   PARENTS    C-FUNCTION
     HAS-COMMENT
     HAS-NAME                     td_onddgs
     HAS-NUM-CALLS-GLOBAL         1
     HAS-NUM-CALLS-TYPEDEF        1
     HAS-NUM-CALLS-MACRO          2
     HAS-NUM-DEFINED-IN           1
     HAS-NUM-REFERENCED-BY        2
```

Figure 3: the CODE-BASE interface

The two upper panels allow the user to browse the concept structure and examine individual concepts. In the upper left is a scrollable graph of the concept taxonomy. When the user points to a particular concept, that concept definition appears in the upper right. This definition includes its parent concepts, children concepts, roles and constraints, and also information about how many individuals of that concept exist and how many have been loaded from the database.

In the middle of interface is a query panel that allows the user to type a CODE-BASE query. After the query is typed, the set of matching instances is displayed in the lower left instance-list panel. Pointing to one of these instances causes CODE-BASE to display that instance in the lower right instance panel. This display includes the instance name, its parent concepts, slot information, and lists of any slot instances which have been loaded. When the user points to various slot descriptions of an individual, this is interpreted by CODE-BASE as a request to load all the fillers of that slot from the database and display them in the instance-list panel (this could also be accomplished by a CODE-BASE query). By alternately selecting an instance in the instance-list and a role in the instance panel, the user can navigate through the network of individuals in a "hypertext"-like fashion. If the user selects the "has-path-name" role of a file, an editing window is opened up so the user can examine the source file itself.

3. Using CODE-BASE for Discovery

This section has two parts. The first briefly describes a discovery session where new concepts are created, added to the taxonomy, and used in the creation of other concepts. It describes the queries the user might generate, the responses to those queries, and, at the end of the section, illustrates what the concept taxonomy would then look like. The second section discusses the role of classification in the discovery process.

3.1 A Discovery Scenario

Assume that a software developer is interested in understanding how the "last-number-dialed" (LND) feature is implemented, as briefly described in the introduction. The user knows that this feature is implemented primarily in the "call process" but also sends messages to other processes in the switch. The user is going to take advantage of the assumed existence of naming conventions, as well as the existence of a structured comment within each source file. This scenario is, by necessity, an overview only.

Looking for a place to start, the user decides to try to locate functions involving dialing in general. The user types the following query into the query pane of the CODE-BASE interface:

define dial-function parent: c-function = {x:c-function | NAMEKEY (x.has-name, "dial") }

This forms a new concept, dial-function, and populates the concept with all functions defined in files with the string "dial" in their names. There are X of them. The user then uses the browsing ability in the interface to examine each function and discovers that the function "lastdial" is the primarily function for this feature. The user then types:

define lastdial-function parent: c-function = {x:c-function | x.has-name = "lastdial" }

which saves this information in an explicit concept as a sub-concept of the previously created **dial-function.** The user opens up an editing window to the defining file of the **lastdial** function and observes that several of the macros and global variables used contain the string "lnd". The user uses this observation to create a class of object called **lnd-object** as shown here:

define lnd-object parent code-object = { x:code-object | NAMEKEY (x.has-name, "lnd") }

Now the user creates two other concepts by anding this concept with the concepts **c-macro** and **c-typedef**:

define lnd-macro = { lnd-object AND c-macro }

define lnd-typedef = { lnd-object AND c-typedef }

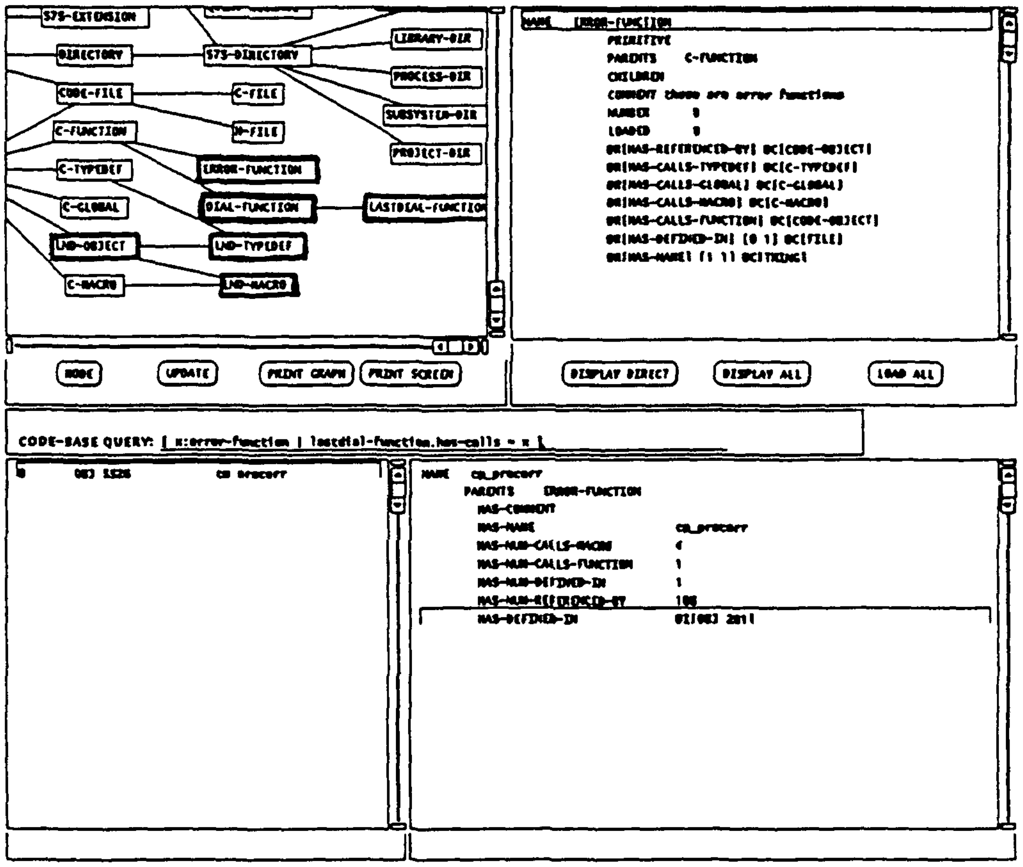
The user has now "discovered" the function which implements the last-number-dialed feature and a number of macros and typedefs which are also involved. Now the user searches for error functions by matching the string "err" to either the name of a function or the comment line for a file:

define error-function parent:c-function = { x:c-function | NAMEKEY (x.has-name, "err")
    OR KEYMATCH (x.has-defined-in.has-comment, "err") }

Now the user can determine which of these X error functions are called by the **lastdial-function**:

{ x:error-function | lastdial-function.has-calls-function = x }

At this point the concept taxonomy is shown in figure 4, which the added concepts highlighted:



```
NAME    ERROR-FUNCTION
        PRIMITIVE
        PARENTS    C-FUNCTION
        CHILDREN
        COMMENT these are error functions
        NUMBER     0
        LOADED     0
        OR[HAS-REFERENCED-BY] OC[CODE-OBJECT]
        OR[HAS-CALLS-TYPEDEF] OC[C-TYPEDEF]
        OR[HAS-CALLS-GLOBAL] OC[C-GLOBAL]
        OR[HAS-CALLS-MACRO] OC[C-MACRO]
        OR[HAS-CALLS-FUNCTION] OC[CODE-OBJECT]
        OR[HAS-DEFINED-IN] [0 1] OC[FILE]
        OR[HAS-NAME] [1 1] OC[THING]
```

```
CODE-BASE QUERY: { x:error-function | lastdial-function.has-calls = x }
```

```
0       OBJ 1526        cm_procerr
```

```
NAME   cm_procerr
   PARENTS    ERROR-FUNCTION
     HAS-COMMENT
     HAS-NAME                       cm_procerr
     HAS-NUM-CALLS-MACRO            4
     HAS-NUM-CALLS-FUNCTION         1
     HAS-NUM-DEFINED-IN             1
     HAS-NUM-REFERENCED-BY          108
     HAS-DEFINED-IN                 0I[OBJ 2011
```

Figure 4: the results of Discovery

3.2 Classification in Discovery

At this point it is worthwhile describing exactly how classification has supported this discovery scenario. First of all, the concept taxonomy as maintained by Classic makes it very easy to understand and   browse the taxonomy itself. More important, it allows the user to add new concepts to the taxonomy and have them maintained as well. In addition, CODE-BASE takes the query used to define a new concept and incorporates that into a Classic rule. Classic rules are forward-chaining rules which will guarantee that, for example, if a new function is added to the body of code that matches the definition of **error-function**, that new function will be properly classified. Another use of Classification in the above scenario was the use of **anding** two existing concepts to create a third. Finally, since the taxonomy is an inheritance network, queries can make use of any concept and be guaranteed that instances of sub-concepts are automatically the subject of the query; for example, a query could refer to **Ind-object** and that reference would include **Ind-macro** and **Ind-typedef** as well.

4. Discussion

The general approach of building knowledge-based Software Information Systems to aid in discovery and maintenance of large software systems is somewhat complementary to other efforts. For example, it seems clear that the entire software process could benefit from knowledge representation technology [13], and advances in software specification research [23], new languages more amenable to structured programming like Ada and C++, the "apprentice" approach [20], and work in knowledge-based software environments [22] all represent progress in this area. Our work in SIS's is more oriented towards reverse-engineering of existing large systems [12]. It seems reasonable to contemplate that both kinds of efforts would benefit from using the same knowledge representation technology, since systems developed using a new paradigm will still need maintenance. Another important area is the capture and representation of design decisions in a large system, which can be very important to understanding and modifying the implementation [8].

Our specific work in CODE-BASE has concentrated on representing syntactic code knowledge that can be extracted automatically. The design of an appropriate ontology and mechanisms for populating it have been achieved. In addition, a mechanism for demand loading of code knowledge, necessitated by the large amounts of code information generated from even a modest set of source files, has been designed and implemented. Several aspects of Classic make this demand-loading work. Most importantly, Classic was designed to work with partial descriptions, and be fully amenable to adding new information as it comes along. This allows it to work with "object_265", knowing only that it is a function, but nothing more about it until needed. Individuals of this sort are similar to "resident object descriptors" in the ORION system [11] (although the ORION system is dealing with more complicated DB issues). Other convenient aspects of Classic include the ability to add new concepts, add new information to slots, and "close" slots, indicating that there are no more fillers for that slot. In this case Classic does not even call the DS for DB information.

Some aspects of our domain also contribute to the efficacy of this approach. For example, there is no sharing of knowledge by multiple users and users do not enter knowledge that will be used by others (although they may use Classic to form conceptual classes of interest to them). This means that representing different versions of the KB, as done by Mays [17], is not necessary; and integrity checking of new knowledge, which is a key issue in the CYC project [15], also doesn't arise.

5. Conclusion

We have described a project, CODE-BASE, whose goal is to develop a knowledge-base ' Software Information System starting with automatically acquired code information, organ zed in a knowledge representation system that provides automatic classification and inference. Most of the code information is kept in a database and loaded on demand, when a query required it. Further research will expand the representation domains and provide better access to what is represented.

6. References

1. AT&T Technical Journal, Special Issue on the System 75 Digital Communications System, Vol. 64, No. 1, Part 2, January 1985

2. Abarbanel, R.M. and M. D. Williams, A Relational Representation for Knowledge Bases, in: Expert Database Systems, L. Kerschberg, ed., Benjamin-Cummings, 1987.

3. Belanger, D.G., Brachman, R.J., Chen, Y.F., Devanbu, P.T., and P. Selfridge, Progress Towards Software Information Systems, to appear in a Special Issue of the AT&T Technical Journal on Software Productivity, Fall, 1990

4. Bernstein, P.A., Database System Support for Software Engineering - An Extended Abstract, Proceedings of ICSE-9: 166-178, 1987

5. Borgida, A., Brachman, R.J., McGuinness, D.L, and L. A. Resnick, CLASSIC: A Structural Data Model for Objects, Proceedings of the 1989 ACM SIGMOD Int'l. Conf. on Management of Data, 1989

6. Brachman, et al., Living with Classic: When and How to Use a KL-ONE-Like Language, to appear in: Formal Aspects of Semantic Networks, J. Sowa, Ed., Morgan Kauffman, 1990

7. Chen, Y.F., Nishimoto, M, and C. V. Ramamoorthy, The C Information Abstraction System, IEEE Trans. on Software Engineering, March, 1990

8. Conklin, J., Design Rationale and Maintenance, MCC Technical Report No. STP-249-88, 1988

9. Devanbu, P.T., Selfridge, P.G., Ballard, B.W., and R.J. Brachman, Steps Towards a Knowledge-Based Software Information System, Proceedings of IJCAI-89, August, 1989

10. Devanbu, P.T., Brachman, R.J., Selfridge, P.G. and B.W. Ballard, A Classification-Based Software Information System, Proceedings of ICSE-90: 249-261 , 1990

11. Harrison, W.H, Shilling, J.J. and P.F. Sweeny, Good News, Bad News: Experience Building a Software Development Environment Using the Object-Oriented Paradigm, Proceedings of OOPSLA-89: 85-94, 1987

12. IEEE Software, Special Issue on Maintenance, Reverse Engineering, and Design Recovery, January, 1990

13. Kaiser, G.E. and P.H. Feiter, An Architecture for Intelligence Assistance in Software Engineering, Proc. 9th ICSE: 180-188, 1987

14. Kim, et al., Integrating An Object-Oriented Programming System with a Database System, Proceedings of OOPSLA '88: 142-152, 1988

15. Lenet, D.B. and R.V. Guha, "Building Large Knowledge Bases", Addison-Wesley, 1990

16. Litman, D.J. and P. Devanbu, CLASP: A Plan and Scenario Classification System, Internal Memorandum, AT&T Bell Laboratories, 1990

17. Mays, E. et al., A Persistent Store for Large Knowledge Bases, in 6th Conf. on Applications of AI: 169-175, 1990

18. Metaxas, D. and T. Sellis, A Database Implementation for Large Frame-Based Systems, in 2d Int'l. Conf. on Data and Knowledge Engineering for Manufacturing and Engineering: 19-25, 1989

19. Patel-Schneider, P.F., Brachman, R.J. and H.J. Levesque, ARGON: Knowledge Representation meets Information Retrieval, in Proc. First Conference on AI Applications: 280-286, 1984

20. Rich, C. and R.C.Waters, The Programmer's Apprentice, Addison-Wesley, 1990

21. Selfridge, P.G. and R.J. Brachman, Supporting a Knowledge-Based Software Information System with a Large Code Database, position paper for the Knowledge-Base Management Workshop, AAAI-90, Boston, Mass, 1990

22. Smith, D.R., Kotik, G.B. and S.J. Westfold, Research on Knowledge-Based Software Environments at Kestrel Institute, IEEE Trans. on SE: SE-11:1278-1295

23. Wileden, J.C., Wolf, A.L., Rosenblatt, W.R., and P.L. Tarr, Specification Level Interoperability, Proceedings of ICSE-90: 74-85, 1990

A DESIGN-PARADIGM-BASED KBSA ARCHITECTURE

Paul A. Bailes
Ron Weber

Language Design Laboratory
Key Centre for Software Technology
The University of Queensland QLD 4072
AUSTRALIA

Phone: + 61 7 377 2097
Fax: + 61 7 371 0783
E-mail: paul@uqcspe.cs.uq.oz.au

ABSTRACT

Motivations for a new style of KBSA/CASE tool - a Design Genesis Support System (DGSS) - are presented in terms of the blind spots in existing research and development. Both language- and method-independence are some of the benefits that should accrue from addressing that part of the development cycle that precedes the enunciation even of a system specification, formal or otherwise.

A DGSS supports the most abstract design paradigms employed by programmers/designers. The paradigms this system aims to support have been determined by experiments. Originally intended to discern differences in programmer behaviour due to language/education variables, the model of problem-solving developed to register the distinctions is applied in a predictive, synthetic role instead of a mere analytic one.

The prototype Paradigm-Oriented Programming Environment (POPE), both demonstrates how off-the-shelf technologies can be combined for novel forms of programmer support, and by its simplicity manifests good prospects for the incorporation of more sophisticated technologies as they emerge.

INTRODUCTION

Our contribution is an expansion of the potential applicability of KBSA technology to provide support for the initial steps taken by a software system designer when attempting to tease out a detailed (possibly formalised) system architecture from an unstructured informal requirements specification. This facility is achieved through a design tool based upon an empirically-derived problem-solving model. The distinguishing feature of this model is that it eschews dependency upon particular specification/design/programming language couplings, and likewise eschews any exclusive focus on that part of the software development process in which formality (either of methods or of representation) is required. Rather, our model captures the generality of human problem-solving behaviour in the computer software design domain. The generality is sufficient to transcend language boundaries, even to apply to the very process of formalising an as yet informal problem statement, yet informative enough to dictate a (generic) KBSA/CASE tool architecture.

The Bounds of Current KBSA Research

Most of the successful work to date has avoided attention to the basic design processes, and instead concentrated either on the resulting *representations* (e.g. data-flow diagrams etc.), or upon a quite specific view of the process (e.g. refinement of non-executable specifications into executable implementations). We take KIDS [1], the Programmer's Apprentice [2] and some others [3, 4] as exemplars supporting this contention. This concentration is unsurprising considering the justifiable identification of this problem's solution as necessary for the realisation of what might be termed the "Formalist" school's agenda for the development of our craft [5]. Nevertheless, exclusive attention to the area would be unwise, because of the unrealistic problem-solving model that inheres to the formalist approach. (See also Dasgupta [9] for complementary criticisms.)

That is, there is a mistaken view of software development as proceeding in sequence from informal requirements through their formalisation into a mathematical specification then by (hopefully) mere calculation of an implementation. Software development tools based upon this view are deficient in respect of:

- at least, how they lock their users into particular styles of specification and implementation calculation;
- how they fail to support the process *pre* formal specification;
- and most seriously, how they actually impose a temporal separation of the spefication and implementation processes - there is no room for a development process that permits stages at which informal requirement and program code may co-exist. While the arguments in favour of decoupling specification enunciation from implementation calculation are weighty [6], it is arguable if the current state of the art demands KBSA tools that enforce it.

The Role of Abstract Paradigms in KBSA Development

Now, a KBSA is a behavioural phenomenon, intended to enhance human performance. If improperly matched to whatever genuine optimal human behaviour pa: erns exist, a KBSA would be an organisational and economic fiasco. Expanding upon the latter point of the previous section, it is therefore more than risky to adopt the formalists' idealised abstraction of the software development process in the absence of empirical evidence supporting its at least implicit employment by acknowledged expert programmers. (Recall by comparison the legitimate motivation for goto-less "structured programming" through Dijkstra's [7] testimony about the observed looping structure of experts' code.)

None of this is to deny the place of formal methods, but rather to ensure that KBSA architectures permit their employment most profitably in accord with actual likely patterns of use by human experts (and by the rest of us in emulation!). Dhar and Pople [8] show how expert systems (for manufacturing applications, specifically) benefit by their construction atop not the conventional shallow quantitative rule base, but upon an explicit model of the underlying problem-solving processes. What the appropriate model for software development problem-solving is, and how it enhances KBSA design, are the topics of the remainder of this paper. It is however clear at the outset that a broader model than currently inspires KBSA design will automatically address the issues of language independence and extra-formal applicability.

AN ABSTRACT STATE-SPACE MODEL OF SOFTWARE DESIGN

A detailed series of empirical investigations of programmer problem-solving behaviour has been carried out in order to ensure that our CASE tool architecture genuinely matches the design paradigms used by expert software developers.

Experimental Methodology

Our original goal was to attempt to detect what (if any) were the observable behavioural differences between programming performed according the the major alternative proclaimed schools: Procedural/Imperative; Logic; Functional; and Object-oriented. Because the benefits of some of these may not appear obvious to various observers, it was felt that experimental evidence might help clarify matters. It was intended to proceed by protocol analysis of transcripts of programming sessions performed by a selection of subjects representative of the schools, encoding the various problem-solving steps recorded and discerning different encoding patterns or paradigms associated with adherence to a school. Experimental subjects were given a non-trivial programming/problem-solving exercise, with a time limit to ensure that high-level *design* activities would be completed, but avoiding domination of the activity by tedious coding. Subjects were required to vocalise their thought processes for the transcript made of their problem-solving activities.

It emerged that enunciation of a suitable coding scheme to document and classify these utterances was problematic. The extent of our difficulty was made clear with the realisation that the sought-for encoding of a transcript was in fact the trace of the problem-solution-process space performed whilst solving the particular problem. To construct individual traces it would first be necessary to model in detail the solution space itself. However, this realisation presented an unexpected opportunity: the solution space documentation could serve as a high-level specification of an integrated software specification/design/implementation environment. Its immediate applicability would be in the verification of our problem-solving model, by constraining the activities of its users to conform to the model - if any user discomfort were to be reported, that would imply a potential deficiency in the model.

The following model was developed to explain our observations.

Formalising System (Development) as State (Transition)

At any stage of its development, a Software System (SS) is characterised by its Problem Statement (PS): an expression of what function the system is to perform. A Software Development Process (SDP) is a series of Software System States (SSS), representing successive stages of the development of some SS. A SSS is essentially a PS (augmented as documented below): the PS's of successive SSS's have (intensionally) *consistent meanings*, save that later PS's will be more refined than earlier ones. "Refinement" may be considered to be ordered according to formality (when moving from Requirement to Specification) or executability (when moving from Specification to Implementation). For the Design Genesis activity, the former is of course the more significant.

Software System State Space

Our experiments further reveal that at each SSS, the problem solver has access to more information than that apparently contained in PS's, and that PS's are structured in particular ways. Dealing with the structure first, we identify the obvious familiar division of a large software system into components: modules, and data and functional abstractions. Significantly, our experiments show that the identification of components characterised informally, in "Requirements" terms, *co-exists* in SSS's with other components well-developed in "Implementation" terms, an occurrence about which our model appropriately makes no prohibition.

Moreover, our subjects conspicuously focussed their attention upon small parts of the PS during problem-solving, parts characterised either by formal linguistic structures (in Specifications) or natural language phrase structures (in Requirements). Thus, in each SSS, the PS is augmented with a Focus - in effect a subset-designator of the PS.

Another prominent problem-solving concept used by our subjects was memory of preceding events to inform subsequent progress, e.g. the existence of a solution to a similar problem, or simply knowledge of why a particular line of development would have been fruitless under prevailing circumstances. Thus, each SSS also contains the History of problem-solving performed to date.

Now, both PS and History are fuzzy concepts: what a problem-solver understands about the current problem or remembers about the past may not be explicitly documented in an SSS. For example, pre-existing libraries of specification/program components may or may not be regarded as belonging to the current SDP. Ideally, the goal of a DGSS is to extend the capabilities of the human problem-solver beyond those native to him/her. It is natural to view this extension as embracing not just better awareness of his/her prior experiences and knowledge, but also what has happened in the local team, community, profession, species, etc. Conceptually, our model copes with this ideal by ascribing to PS and History perfect knowledge of what resources are available and what has happened in the past. Pragmatically, the quality in general of a DGSS directly relates to the extent to which its implementation approaches the ideal in the depth and accuracy of the information it retains.

State Transitions

How may a SSS differ from its predecessor?

First, the Focus may change - the problem-solver may shift his attention from one part of the PS to another.

Second, the PS may itself be changed (e.g. in refining a Specification or formalising a Requirement). However, there is an important restriction on PS changes: only the PS fragment designated by the current Focus may be transformed, corresponding to the reality that a human problem-solver only changes items to which (s)he is attending.

Third, whenever a new SSS appears, its History is automatically derived: the SDP up to and including its preceding SSS.

Thus DGSS quality is specifically determined by the support it offers for state changes of these natures.

RELATIONSHIP TO OTHER MODELS

From the super-abundance of literature on problem-solving in general and programming-problem-solving in particular, Aguero's and Dasgupta's [9] Theory of Plausible Designs (TPD) provides the richest counterpart to our model. Indeed, we like to view ours as a refinement of the synthesis-directed facets of TPD. Like ours, TPD provides a uniform structure for problem solution knowledge, so avoiding *a priori* distinctions between requirements, designs and programs. Indeed, our abstract model as fully-developed [10] is sufficiently general to cope with developments in programmers' problem-solving expertise, by recording in the PS the paradigms accessible to programmers.

The most significant contributions we make are in the area of modularity. The Focus concept, and how refinements are restricted to a nominated focus, naturally lead into the imposition of a hierarchical structure upon the solution as it is developed through successive SSS's. Modularisation is further supported by how the concrete KBSA tool we develop in consequence (as exposed below) instantiates our model's hitherto abstract History concept.

A CONCRETE MODEL FOR KBSA DESIGN

In re-iteration, KBSA quality is determined by the extent to which it supports desired knowledge representations and relevant transformations.

Reducing Abstraction

The abstract model can be, and in fact requires, elaboration for its structures and operations to match actual software development situations. That many such elaborations are possible for different situations signifies the utility of the model. For example, the various PS's may be written in any language. The means of executing refinement, or a discipline thereof that may be glorified with the title "Programming Method(ology)", are likewise left open. At least, a PS may be viewed as being represented by unstructured text, and changes to a PS effected by straightforward text editing. Indeed, it is important to present our prototype, experimental DGSS in this way to avoid confusing its success (or failure) with that of a particular flavour of specification/programming language and associated tool set.

The existing treatment of the History SSS component needs special attention, because of the effective inaccessibility of chosen components. That is, one of the main uses of the SDP History is for backtracking - trying alternative development paths should the current prove fruitless for some reason.

The remainder of this section outlines how a refined view of History affects the abstract model, and how one concrete elaboration takes form.

Partitioned SDPs

Now, there is no reason why backtracking should involve the entire SSS, but just that part concerned with that component of the system requiring re-working. Therefore, we re-present the entire SDP as a collection of individual SDPs, the History component of each SSS concerning the previous states of the SDP to which it belongs only. Consequently, SDP representations are required that allow the multiple SDP's to co-exist, to refer to each other (corresponding to references by one component of an SS to another), and to be brought into being on demand (as the need for the relevant component is identified in the parent SDP). Serendipitously, this partitioning accommodates multi-person, team efforts and also re-use of SDP's developed elsewhere.

Without this partitioning, the History facility would be useless. For example, if refinements of two foci in successive PS's occurred in alternation (which might just happen to suit the problem-solver's *modus operandi*), it would be difficult to restore an early version of one of the foci without restoring the other. While all the necessary information for selective restoration only is available in the abstract, unstructured/monolithic view of History, access procedures remain unspecificed. Partitioning History seems to provide appropriate such access procedures implicitly.

In our prototype implementation, each SDP is allocated its own window in a stock workstation environment (SUN Microsystems' SunView). There is maintained a current SDP state, and the sequence of prior states for that window. To allow complete linguistic independence, the windows invoke the standard plain text editor, applied to files named to identify the SDP. From the names are derived file names for both the current SDP state, and the SDP history.

Concrete Transitions

The resulting system we call POPE (the Paradigm-Oriented Programming Environment), though the name is misleading in that (i) it most definitely applies to aspects of the *software* life cycle other than "mere" programming, and (ii) the supported Paradigms are most definitely not the language-specific kind alluded to by Floyd [11], but reflections of the most abstract language-independent strategies we have been capable of discerning.

A POPE *session* commences by invoking the editor on a new window and entering the problem description. This text can be thought of as the PS of the initial state of the SDP. Subsequent operations on the window (invoked by mousing) are selected from the following.

Change Focus

A new part of the current PS is identified as the Focus. In POPE, this is performed by highlighting text with the editor.

Refine Focus

A new (temporary) edit window is opened containing the current focus, which may be changed as desired. Upon termination, the original window's Focus is updated with the changes made. History for the SDP window is also updated.

Backward Restoration of History

The previous window is restored. Successive applications of this option back up through successive prior states.

Forward Restoration of History

The subsequent window is restored. Successive applications of this option advance through successive states (only applicable after Backward Restoration).

Spawn SDP

Corresponding to a top-down, stepwise refinement approach, a part of the PS is recognised of sufficient import as warranting a separate SDP. A new window is added, named from the text of the current Focus. Alternatively, a new (abbreviated) name can be provided, with the full text of the Focus becoming the initial text of the new edit window. In either case, the Focus of the original window is set to the name established for the new window.

Augment with a new SDP

Corresponding to a bottom-up approach, the need for a new component is realised. A new named window is opened for an initial text description to be entered. When later developments of other SDPs refer to this name, the existence of the named window implies that no new expansions/window establishments need be made.

Other facilities are provided to help manage the amount of detail exposed and to establish links with other POPE sessions.

Movement from SDP window to SDP window is effected by basic window selection operations. SDP sessions can be saved and restored. The conclusion of a series of SDP sessions is a collection of named files storing the source code of the modules comprising a SS.

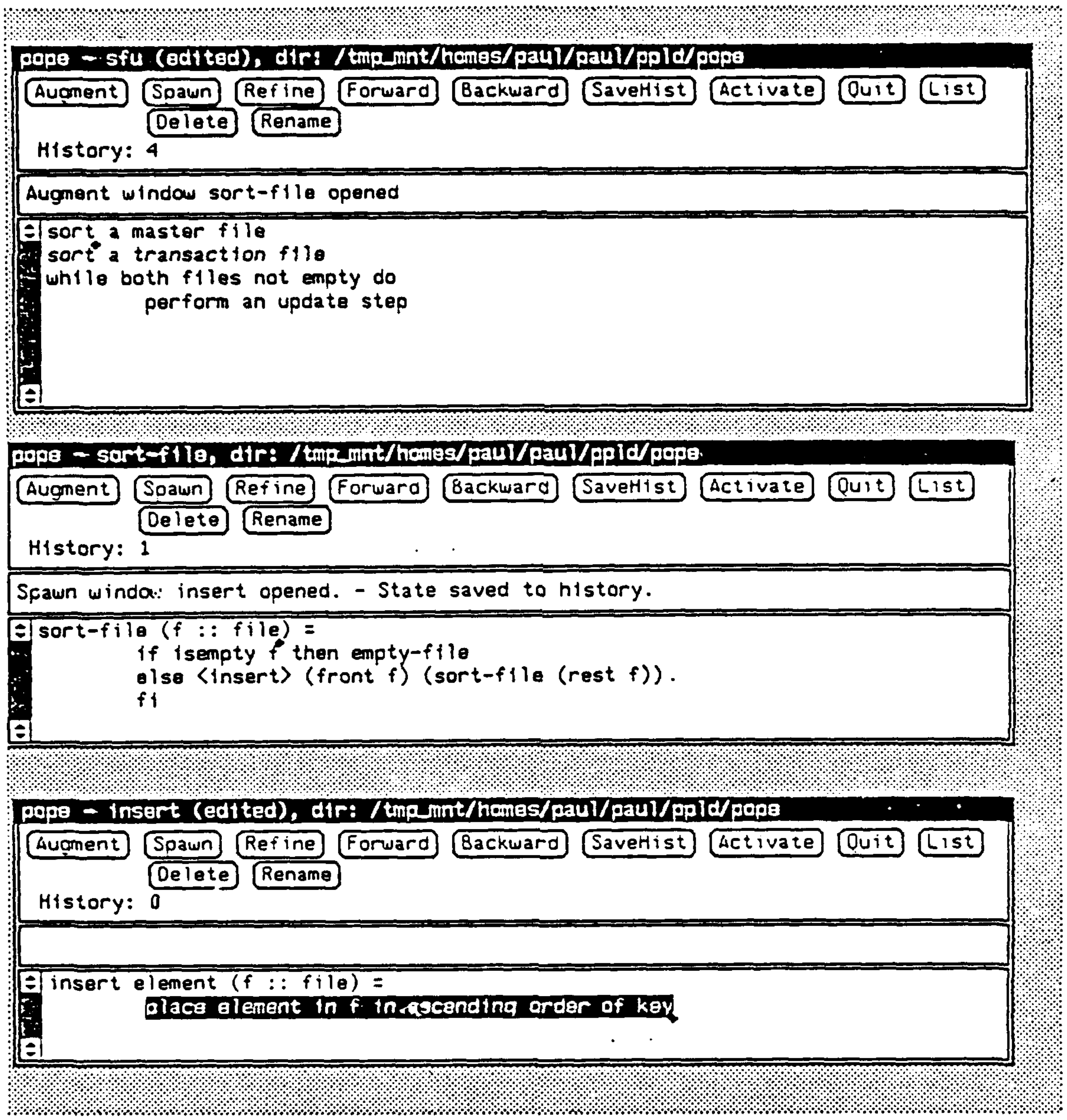
EXAMPLE

Figure 1 shows an SS consisting of three SDP windows. The top window is the SDP for the "top level" of the development of a sequential file update program. Several refinements have been already performed on its PS (the "History: 4" message). The most recent development of "sfu" has been the augmentation by a new SDP "sort-file" to define the process of sorting a data file. As yet, the relationship between "sort-file" and "sfu" is yet to be made explicit - presumably some future refinement of the latter will refine the "sort a ... file" lines into versions referring to "sort-file". The "sort-file" SDP has used a formal notation (modelled on a modern functional-style programming language) from the outset for its PS, incidentally. From "sort-file" has been spawned the "insert" SDP, which is defining the "insert" function used in "sort-file". This definition is presented with an informal body, which is designated as the Focus, just about to be refined.

Figure 2 depicts the change effected to figure 1 by selecting "Refine" from the "insert" SDP menu. A new temporary window is opened with the text of the Focus, upon which may be performed arbitrary edits. When "EndRefine" is selected, the highlighted text in "insert" will be replaced by the (changed) contents of the temporary window, which then quits.

Figure 3 shows that general SunView facilities are available to manage screens. Augmenting the SS with an SDP for the "file" abstract data type crowds the picture, so "sort-file" and "insert" have been shrunk to icons.

Figure 4 indicates spawning under way. Assuming a functional language as implementation target, we can package tail-end recursion as a familiar while-loop, and so desire to provide a separate, named facility for this concept. We proceeded by selecting the "while" word as the Focus in the "sfu" window, and selecting "Spawn" from the menu. The system response was to indicate that "while" now named an SDP (by enclosure in "< ... >", see also "insert" in the "sort-file" window of figure 1), and to open a new "while" window with its name as the default text (facilitating a likely behaviour of having the SDP provide a definition for the object named by the window).



pope - sfu (edited), dir: /tmp_mnt/homes/paul/paul/ppld/pope
Augment Spawn Refine Forward Backward SaveHist Activate Quit List Delete Rename
History: 4

```
Augment window sort-file opened
```

```
sort a master file
sort a transaction file
while both files not empty do
        perform an update step
```

pope - sort-file, dir: /tmp_mnt/homes/paul/paul/ppld/pope
Augment Spawn Refine Forward Backward SaveHist Activate Quit List Delete Rename
History: 1

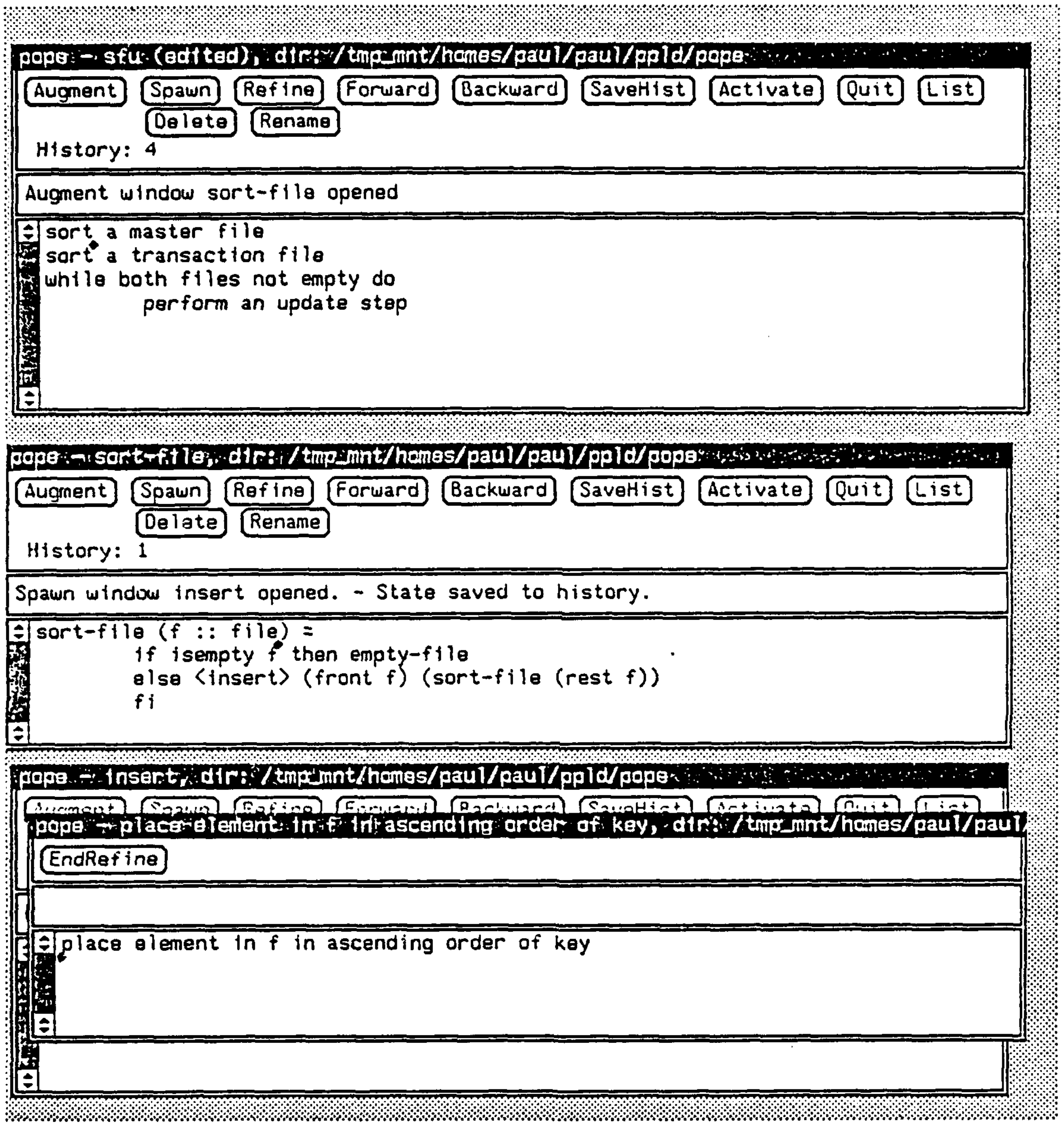
```
Spawn window insert opened. - State saved to history.
```

```
sort-file (f :: file) =
        if isempty f then empty-file
        else <insert> (front f) (sort-file (rest f)).
        fi
```

pope - insert (edited), dir: /tmp_mnt/homes/paul/paul/ppld/pope
Augment Spawn Refine Forward Backward SaveHist Activate Quit List Delete Rename
History: 0

```
insert element (f :: file) =
        place element in f in ascending order of key
```

Figure 1: Ready to refine


pope - sfu (edited), dir: /tmp_mnt/homes/paul/paul/ppld/pope
Augment
Spawn
Refine
Forward
Backward
SaveHist
Activate
Quit
List
Delete
Rename
History: 4

```
Augment window sort-file opened
```

```
sort a master file
sort a transaction file
while both files not empty do
        perform an update step
```

pope - sort-file, dir: /tmp_mnt/homes/paul/paul/ppld/pope
Augment
Spawn
Refine
Forward
Backward
SaveHist
Activate
Quit
List
Delete
Rename
History: 1

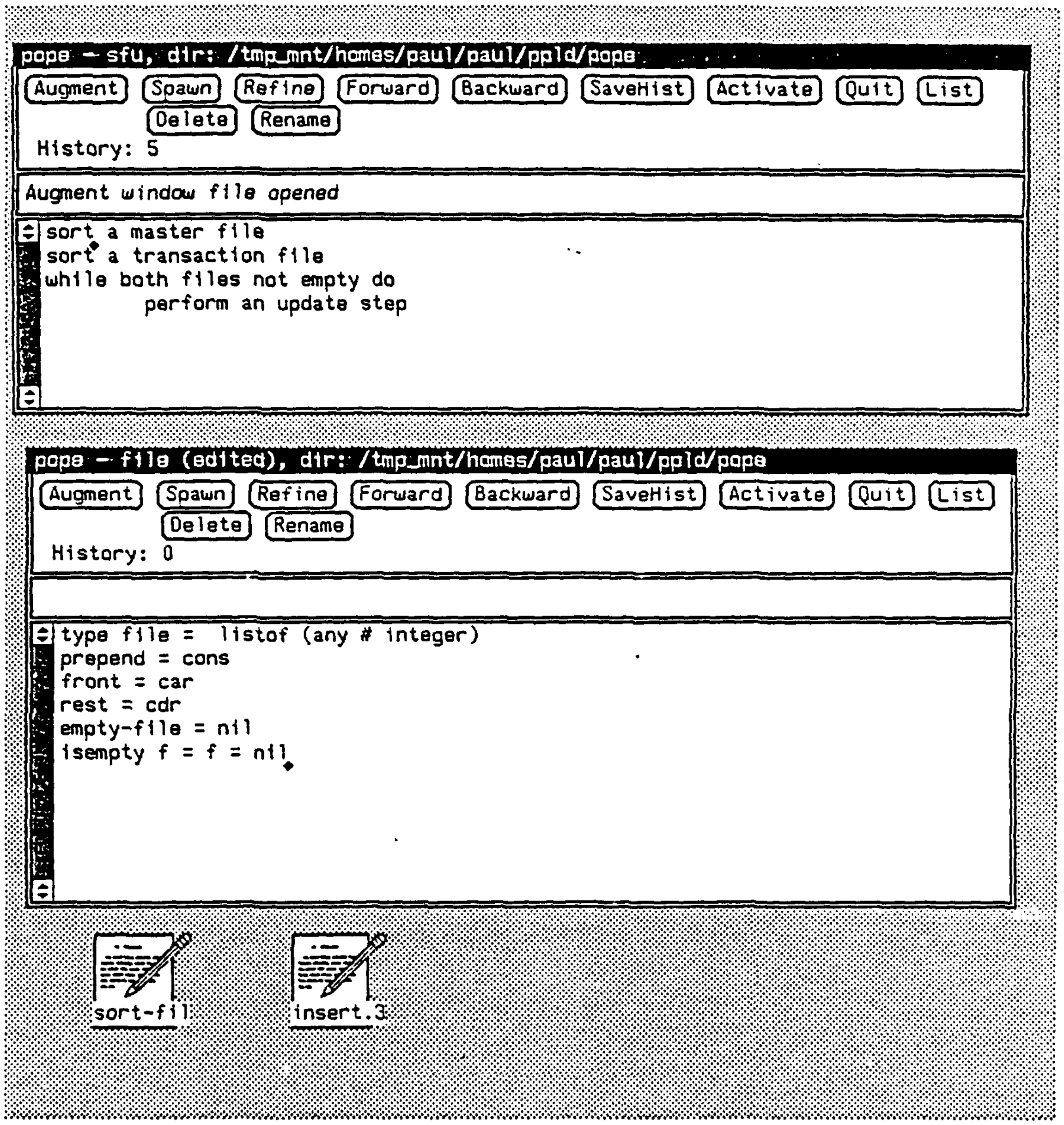
```
Spawn window insert opened. - State saved to history.
```

```
sort-file (f :: file) =
        if isempty f then empty-file
        else <insert> (front f) (sort-file (rest f))
        fi
```

pope - insert, dir: /tmp_mnt/homes/paul/paul/ppld/pope
pope - place element in f in ascending order of key, dir: /tmp_mnt/homes/paul/paul
EndRefine

```
place element in f in ascending order of key
```

Figure 2: Starting a refinement

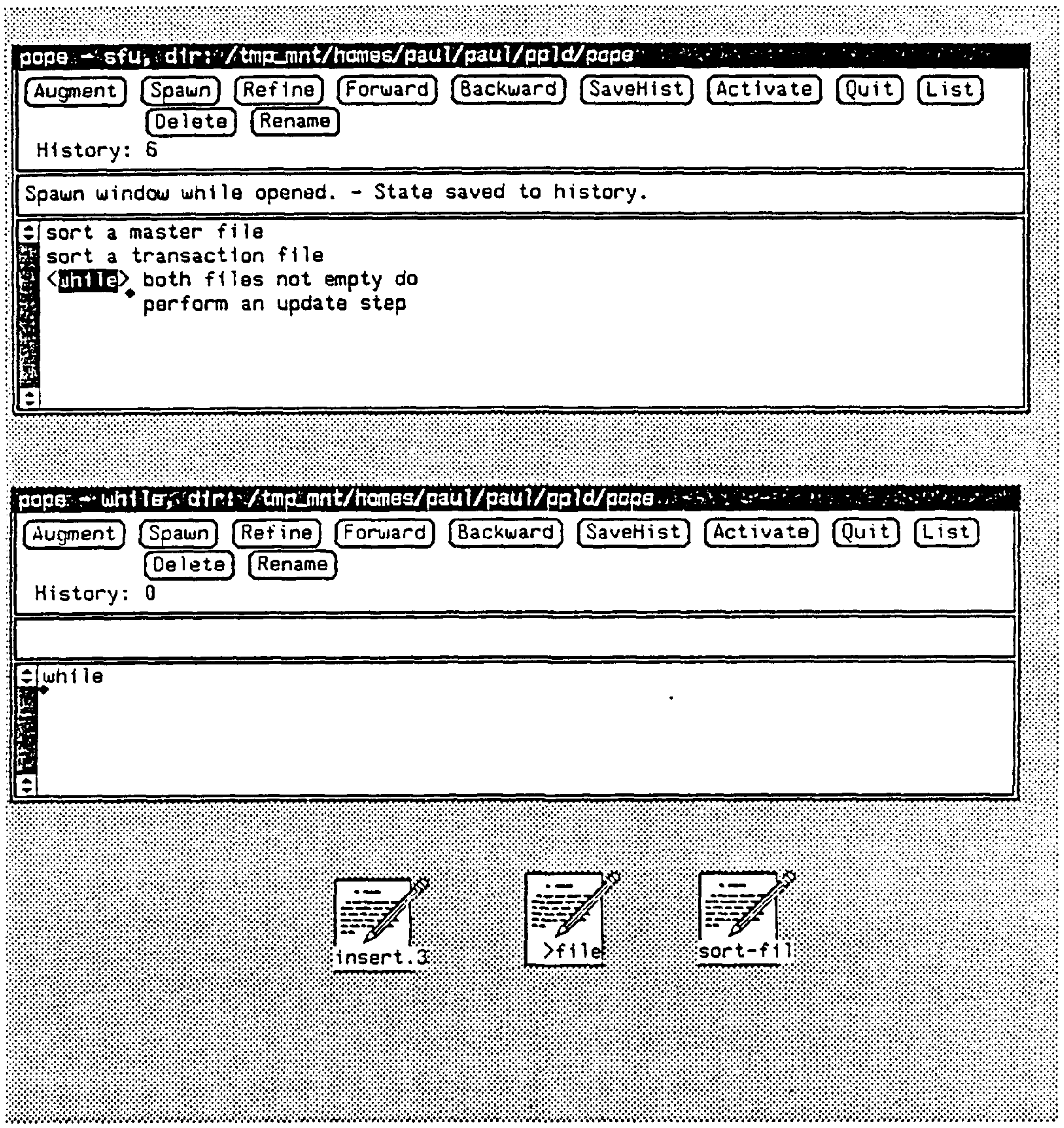


Augment window file opened

```
sort a master file
sort a transaction file
while both files not empty do
        perform an update step
```

```
type file =  listof (any # integer)
prepend = cons
front = car
rest = cdr
empty-file = nil
isempty f = f = nil
```

Figure 3: A tidy screen



```
Spawn window while opened. - State saved to history.
```

```
sort a master file
sort a transaction file
<while> both files not empty do
        perform an update step
```

```
while
```

Figure 4: Starting to spawn

DEVELOPMENT POTENTIAL

The POPE prototype DGSS represents one of the initial stages of a project aiming to provide an integrated Ada-oriented development environment, from specification development to detailed designs initially, and ultimately to implementations using transformation technology [1]. The first task is to validate POPE by adequate trials. Complementing the enhancements to POPE foreshadowed below are an Ada-oriented formal design and specification language and a generic, multi-language editor.

An Ada-oriented specification/design language, drawing upon Luckham et al's ANNA [12] and Lees' ADL [13] will be designed to bridge the gap between the process of devising a formal specification of a software system and its implementation in an executable programming language, i.e., integration of program specification, design and implementation languages.

Sophisticated Design Structures And Powerful Processing Tools

The partitioned, hierarchical SSS structure, and the links between its elements, are evocative of an Hypertext [14] document. The prototype DGSS will be re-implemented atop a Hypertext base, and opportunities taken for design improvements as revealed by experience and in the context of the more powerful implementation technology.

Multi-language SSS Structures

While our CASE tool architecture is language-independent, and while the problem-solving strategies it supports can be promoted to contexts that intermingle Requirements, Specifications and Implementations, this flexibility should be accommodated with the reality of the current existence of powerful language-specific CASE tools. This composition is feasible because the language-specific tools operate in areas orthogonal to *general* problem-solving, such as text entry, display and analysis. The project's ultimate commitment to supporting the genesis of designs in a specific design/specification language makes the addressing of this issue all the more urgent.

Fortunately, the UQ2 generic editor [15] intended to provide the basic such capability to the project supports a rudimentary, though fully multi-language, document hierarchy structure. Therefore, the complete realisation of the DGSS potential will be represented by a UQ2-derivative, instantiated for particular design languages of our choice *and* enhanced to support Hypertext relationships between the components of a complete document.

CONCLUSIONS

Our KBSA architecture has the following merits.

(1) It is philosophically appealing, in that its goal is to augment human problem-solving resources toward the level of perfect knowledge of relevant experiences and tools across a community of individuals.

(2) It is based not on our introspective theoretical fantasies but on the need to find a model matching observed human behaviour.

(3) Its technological neutrality allows it to be employed either in a low-technology environment (as a shell to organise the invocation of simple text preparation tools) or to inspire the design of KBSAs that address the "hard" technology of specification refinement. A prototype of the former arrangement has been crafted using standard workstation tools; the latter we hope to achieve as part of an environment for an Ada-oriented design and specification language.

ACKNOWLEDGEMENTS

We are grateful to Andrew Lister for his collaboration, and to Eric Halil for programming and research support as well.

This research has been supported by the Australian Research Council, and continues under the auspices of the Advanced Software Engineering Environments Consortium sponsored by the Australian Defence Science and Technology Organisation.

REFERENCES

1. Smith, D.R., "KIDS - A Knowledge-Based Software Development System", KES.U.88.7, Kestrel Institute (1988).
2. Rich, C. and Waters, R.C., "The Programmer's Apprentice - A Research Overview", IEEE Computer (November, 1988).
3. Heisel, M., Reff, W. and Stephen, W., "Formal Software Development in the KIV-System", Proceedings of the Workshop on Automating Software Design, IJCAI-89, pp. 115-121 (1989).
4. Mostow, J., "Exploiting DIOGENES' Representations for Search Algorithms", Proceedings of the Workshop on Automating Software Design, IJCAI-89, pp. 187-200 (1989).
5. Hoare, C.A.R., "An Overview of Some Formal Methods for Program Design", IEEE Computer, vol. 20, no. 9, pp. 85-91 (September, 1987).
6. Hayes, I.J. (ed.), "Specification Case Studies", Prentice-Hall (1986).
7. Dijkstra, E.W., "GOTO Statement Considered Harmful", CACM vol. 11 no. 3 pp. 147-148 (1968).
8. Dhar, V. and Pople, H.E., "Rule-Based versus Structure-Based Models for Explaining and Generating Expert Behaviour", CACM vol. 30, no. 6, pp. 542-555 (1987).
9. Dasgupta, S., "The Structure of Design Processes", in M.C. Yovits (ed.), Advances in Computers, vol. 28, pp. 1-67, Academic Press, San Diego (1989).
10. Bailes, P.A., Lister, A.M. and Weber, R., "A State-Space Model of Some Cognitive Aspects of Software Design", *submitted* for publication.
11. Floyd, R.W., "The Paradigms of Programming", CACM vol. 22 no. 8 pp. 455-460 (1979).
12. Luckham, E., von Henke, F., Krieg-Bruckner, B. and Owe, O., "ANNA: A Language for Annotating Ada Programs", Lecture Notes in Computer Science, vol. 260, Springer Verlag, Berlin (1987).
13. Lees, R.A., "A Tailored Design Language: Putting Model Based Formal Specification into Practice", Proceedings 5th Australian Software Engineering Conference, pp. 159-164, Sydney (1990).
14. Conklin, E., "Hypertext: An *introduction and survey*", IEEE Computer, vol. 20, no. 9, pp. 17-41 (1987).
15. Broom, B., Welsh, J. and Wildman, L., "UQ2: a multilingual document editor", Proceedings 5th Australian Software Engineering Conference, pp. 289-294, Sydney (1990).

The Concept Demonstration Rapid Prototype System[1]

Michael DeBellis
Center for Strategic Technology Research
Andersen Consulting
100 South Wacker Drive, 9th Floor
Chicago, IL 60606
(312) 507-6530
debellis@andersen.com

Abstract

This paper describes the KBSA Concept Demonstration Rapid Prototype system. We present the high level goals of the KBSA Concept Demonstration System. We describe the functionality which exists in the prototype Demonstration System and provide some examples of the system in use. The functionality which is planned for future versions of the Demonstration System is also described.

1. The Concept Demonstration System

An important goal of the KBSA Concept Demonstration project is to communicate the KBSA approach to software development through creation of a Demonstration System. In order to achieve this, we determined that the Demonstration System should:

- Show the full range of KBSA functionality, from gathering of informal requirements to generation of efficient code.
- Emphasize functionality which differentiates KBSA from conventional CASE tools.
- Address realistic problems in domains which are relevant to industrial practitioners.
- Provide a usable and friendly user interface.
- Contain pre-defined scenarios which exhibit the previous characteristics and can be run by non-technical users.
- Be robust and powerful enough to be used by more technical users in an experimental mode independent of scenarios.

1. This research is supported by the Rome Air Development Center (RADC), contract F30602-89-0160. Views and conclusions contained in this paper are the Author's and should not be interpreted as representing the official opinion or policy of RADC, the U.S. Government, or any person or agency connected with them.

As the first step in creating such a system we have developed a Rapid Prototype which defines our approach to integrating KBSA concepts and software and provides a foundation on which to build the Demonstration System. Our approach is to integrate software from other KBSA projects when feasible. When integrating software is not feasible we reimplement functionality from previous projects. We also develop new functionality which we deem necessary to achieve the goals described above.

Our development environment is Refine running on Sun Sparcstations. Because of this, we decided that for the most part we would reimplement functionality from KBSA projects which did not use Refine and integrate KBSA software built with Refine. We decided to reimplement functionality from the Specification Assistant (SpecA) [Johnson 88], Knowledge Based Requirements Assistant (KBRA) [Harris 88], and Aries [Johnson 90a] projects; and to integrate software developed in the Project Management Assistant (PMA) [Gilham 86], the Performance Estimation Assistant (PEA) [Blaine 88], and the Development Assistant [Smith 90] projects.

2.0 Rapid Prototype Functionality

In planning the Rapid Prototype system, we decided to make the reimplementation of requirements and specification functionality the main focus of our effort. There were two main reasons for this decision:

1) The requirements and specification functionality of KBSA are of particular importance to the communication and demonstration goals of our project.

2) All Refine based KBSA software was still under development at the beginning of our project (September 1989). The PMA was under development until July 1990. The PEA is part of the larger Kestrel Interactive Development System (KIDS) which is currently being extended as part of the Development Assistant project.

The following sections describe requirements and specification functionality which we have developed in the Rapid Prototype. Section 2.1 describes functionality we have reimplemented that was demonstrated in the KBRA. Section 2.2 describes functionality reimplemented and integrated from the SpecA and Aries projects. Section 2.3 describes new functionality related to activity coordination which we have implemented to demonstrate the *Assistant* aspect of KBSA. Section 2.4 describes a brief scenario that demonstrates some of this functionality.

2.1 Concepts Demonstrated in KBRA

We have reimplemented two important concepts from KBRA: a presentation based interface and automated assistance in organizing and formalizing informal text strings.

2.1.1 Presentation Based Interface

The use of a *Presentation Based* user interface is a key concept of KBRA. A presentation based interface enables multiple presentations of a knowledge base (KB) in order to provide different, consistent perspectives on editing and viewing that knowledge base. In order to do this the presentation interface must provide the following capabilities:

- A model of how the KB structure translates into each presentation.

- Editing options on each presentation specific to the particular view afforded by that presentation.

- A mechanism for propagating changes made to the knowledge base to presentations which are effected by those changes.

In the Rapid Prototype, the most important use of the knowledge base is to represent requirements and specifications (this is discussed in detail in section 2.2). The Rapid Prototype interface contains the following presentations[2]:

Presentations for Class and Slot Definitions

- Entity Relation (E/R) Diagram. This takes class definitions and translates them to E/R representations. Our mapping between KB classes and the E/R model is: classes are treated as entities, slots with a range that is not a KB class (e.g., integer, string, symbol, ...) are treated as attributes, slots with a KB class range are treated as relations.

- Class Tree. This graphs the subclass links between a group of classes.

Presentations for Instances. These are useful for viewing instances created as a result of executing simulations. Since elements of specifications are also KB instances, e.g., a class definition is itself an instance of the Class (meta) class, these presentations are also very useful for viewing specification objects and the relations between them.

- Object Frame Display. This displays slots for a particular object and the values for the slots. The slots and the slot values are mouse sensitive providing the developer with access to various options such as displaying the slot or value using a presentation, paraphrasing it (see 2.2.3), invoking evolution transformations (see 2.2.2), etc. The developer can control which slots will be seen for

2. Presentation names have the following conventions: *Diagram* presentations show network graphs with no automatic layout. *Tree* presentations provide automatic layout for trees or lattices. *Display* presentations involve mouse sensitive text with no graphics.

instances of each class.

- Semantic Net Diagram. This takes a group of instances and a group of relations and shows the links between those instances. This is especially useful for showing links between groups of hypertext nodes (described in 2.1.2) as well as links from hypertext nodes to formal objects (see figure 3 in section 2.4.3).

Process Related Presentations

- Information Flow Diagram. This provides information on how the slots of a class are accessed and modified by functions and other process oriented specification constructs.

- Syntax Based Display. This displays the specification language (described in 2.2) description for a KB object and performs mouse sensitive highlighting based on the parse tree of the object.

Miscellaneous Presentations

- Knowledge Base Module Tree. This displays the use relations between a group of knowledge base modules (KB modules are discussed in 2.2).

- History Tree. This displays a graph of the actions performed in a development session.

- Hypertext Display. This integrates the Object Frame display with a mechanism for dynamically creating mouse sensitive highlighting for references to objects in text strings (see 2.1.2).

2.1.2 Hypertext Requirements: *Catch as Catch Can*

An important concept in the KBRA is that during development and formalization of requirements, the system should allow the developer to enter requirements in an informal manner (e.g., text) and should provide support for organizing and formalizing these requirements. An example of automated assistance in organizing informal requirements is to have the system recognize words or phrases in text strings that relate to formal objects. This recognition is described with the phrase *Catch as Catch Can*, since the tool will often not recognize words or phrases that relate to formal objects because of the ambiguity inherent in natural language.

The Rapid Prototype uses the Hypertext presentation to enter and view informal requirements. Each requirement is entered as a text string with pointers to other objects such as: the author of the requirement, other requirements with which it may conflict, requirements documents that the requirement is part of, etc. The text string for the hypertext requirement is a *Hyperstring*. A hyperstring and its pointers to other objects is a *Hypertext Node*. When a hyperstring is entered or modified, any words in the hyperstring that correspond to the names of objects describing requirements or specifications for that system are highlighted in bold and are made mouse sensitive, providing the developer with access to various options such as displaying the object using a presentation, paraphrasing it (see 2.2.3), invoking evolution transformations (see 2.2.2), etc. In addition, whenever the knowledge base is updated, part of the job of the presentation interface is to update the display of viewable hyperstrings so that references to objects that have been added or deleted will be updated.

The Rapid Prototype does not use natural language processing techniques, such as those found in KBRA. It is limited to simply processing each word in the string and looking it up in the knowledge base. However, it provides general hypertext functionality that was not present in KBRA, as well as classes of hypertext nodes and hypertext links that are geared toward requirements. These include all the classes (issues, positions and arguments) and links (responds-to, objects-to, supports, ...) in the IBIS [Conklin 88] methodology.

2.2 Concepts Demonstrated in SpecA and Aries

2.2.1 High Level Specification Language

The Gist specification language is an essential component of the SpecA and Aries projects. Gist is a general purpose specification language that allows developers to describe requirements, domains, and systems in concise statements with minimal commitment to implementation decisions. There is a high degree of overlap between Gist and the Refine language. Both Gist and Refine have the equivalent of classes and subclasses, computed values for slots, and logical expressions. In addition, the version of Gist developed for the Aries project [Johnson 90] provides a *Refined-Gist* representation which allows most Gist constructs to be stated in a syntax very close to that used by Refine. We determined that the most important Gist constructs not present in Refine are:

- A general relation construct. Gist allows unary, binary, and n-ary relations as well as a wildcard facility to retrieve tuples matching a particular pattern.

- Constraints. Gist contains constructs for representing logical expressions indicating "X should always be true" (INVARIANTS) and "if X should ever occur, do Y" (DEMONS). Gist provides a construct to indicate that a group of expressions should be executed atomically. We consider these constructs together because they all could be implemented by a constraint mechanism such as that found in AP5 [Cohen 85].

- Time Constructs. Gist contains expressions for representing notions of time such as "every N minutes" and "time since X last occurred".

- Folders. Aries Gist contains a *folder* construct for organizing specification objects into meaningful groups and for describing communication between those groups.

Our strategy for developing a Concept Demonstration specification language is to define an Extended Refine Specification Language (ERSLa[3]) using the Refine grammar definition tools. ERSLa starts with the productions of the Refine language and adds new productions for Gist-like constructs. ERSLa also has transformations which are added to the Refine compiler in order to transform the ERSLa extensions into executable code. Thus, all ERSLa specifications are executable.

The INVARIANT and DEMON constructs, the ANY construct for non-determinism, and folders (which we call KB Modules) are implemented in the Rapid Prototype. These constructs were the most essential to the scenarios that we wished to demonstrate. We intend on implementing most or all of the other constructs described above in future work.

2.2.2 Evolution Transformations and History Mechanism

The SpecA and Aries projects developed the concept of *Evolution Transformations* for performing "stereotypical, meaningful changes to the specification" [Johnson 88] in "systematic, controlled ways" [Johnson 90b]. Evolution transformations are performed with a history mechanism that allows a developer to explore alternative development options and to undo and redo any sequence of development steps.

We have reimplemented those evolution transformations from the SpecA that were best suited to highlighting the difference between KBSA and conventional CASE and knowledge base development tools. These include the Add Parameter, Bundle, Invert Relation, Splice Communicator, and Install Protocol transformations. We have also reimplemented simpler transformations for creating, modifying and deleting specification constructs (e.g., Add/Remove Subclass, Relation, Function, Invariant, ...; Merge/Splice Classes; etc.). A full description of all the Rapid Prototype evolution transformations is available in [Williams 90b].

Our history mechanism creates a new development step for each invocation of an Evolution Transformation. A developer can view a graph of the current development steps and can return a specification to any step in the development history by selecting it from the graph. The history mechanism utilizes the presentation interface. The same facilities that are used to update presentations after a new development step has been initiated are used to make the presentations consistent with a state moved to in the development history.

3. Pronounced: Ursula

2.2.3 Specification Feedback

One advantage of a formal specification is the ability to reason about the specification and provide feedback to the developer. The Rapid Prototype provides three types of specification feedback: Static Analysis, Resource Analysis, and Paraphrasing.

- Static Analysis provides feedback on the syntactic correctness of each specification construct. This includes problems such as type inconsistencies, references to undeclared named objects, and parameter mismatches.

- Resource Analysis provides feedback on which processing objects (functions, invariants, demons, ...) use and/or modify which data objects (classes, slots, global variables, ...).

- Paraphrasing is used to translate formal specification constructs into English. The Rapid Prototype paraphraser is an adaptation of the paraphraser used by the Aries project to paraphrase Gist specifications. See [Williams 90a] for a detailed description of our paraphraser.

2.3 The Agenda Mechanism

An important aspect of KBSA that has not received much attention in previous research is that it is meant to be an intelligent assistant that provides guidance to the developer. In order to demonstrate this aspect we have begun development of an *Agenda* mechanism in the rapid prototype. The agenda provides a list of activities that need to be completed as well as (when possible) suggestions as to how to go about these activities. Agenda items can be created in three ways:

1) Creation by the system as a result of analysis of the current state of the specification.

2) Creation by the system as a result of project management activity.

3) Creation by the developer as a reminder of activities that need to be performed in the future.

The agenda is dynamically updated so that if the developer resolves a problem that is on the agenda it is automatically removed. There is also a model of how various types of agenda items interact, since it will often be the case that resolving one type of problem will automatically resolve a number of other problems as well. This model is used to prioritize the agenda.

2.4 Example Scenarios

The following example demonstrates some of the Rapid Prototype functionality. Figure 1 shows the Rapid Prototype interface. Across the top of the interface is a control panel with options for the history mechanism, knowledge base modules, user interface, and the agenda mechanism. Next to the control panel is a small window for temporary messages. The style of the interface is similar to AI development environments such as KEE [Fikes 85]. All windows can be moved, reshaped, buried, etc. Most user actions are initiated by mousing on an object and choosing options from a pop-up menu. This allows the presented options to be tailored to the object as well as the context in which the object is found (e.g., there will be different options presented for a class object when it is in an E/R diagram than when it is in a class tree).

2.4.1 Initial Display of Classes, Relations and Function Specifications

In figure 1, the developer is viewing four presentations. Directly below the control panel is an information flow diagram viewing the AIRPORT and AIRCRAFT classes and the DETERMINE LENGTH OF FLIGHT and REPORT AIRCRAFT LOCATION function specifications. The diagram shows that DETERMINE LENGTH OF FLIGHT accesses the DEPARTURE POINT and PLANNED DESTINATION slots of AIRCRAFT and the AIRPORT LOCATION slot of AIRPORT. It shows that REPORT AIRCRAFT LOCATION accesses the AIRCRAFT LOCATION and AIRCRAFT ID slots of aircraft.

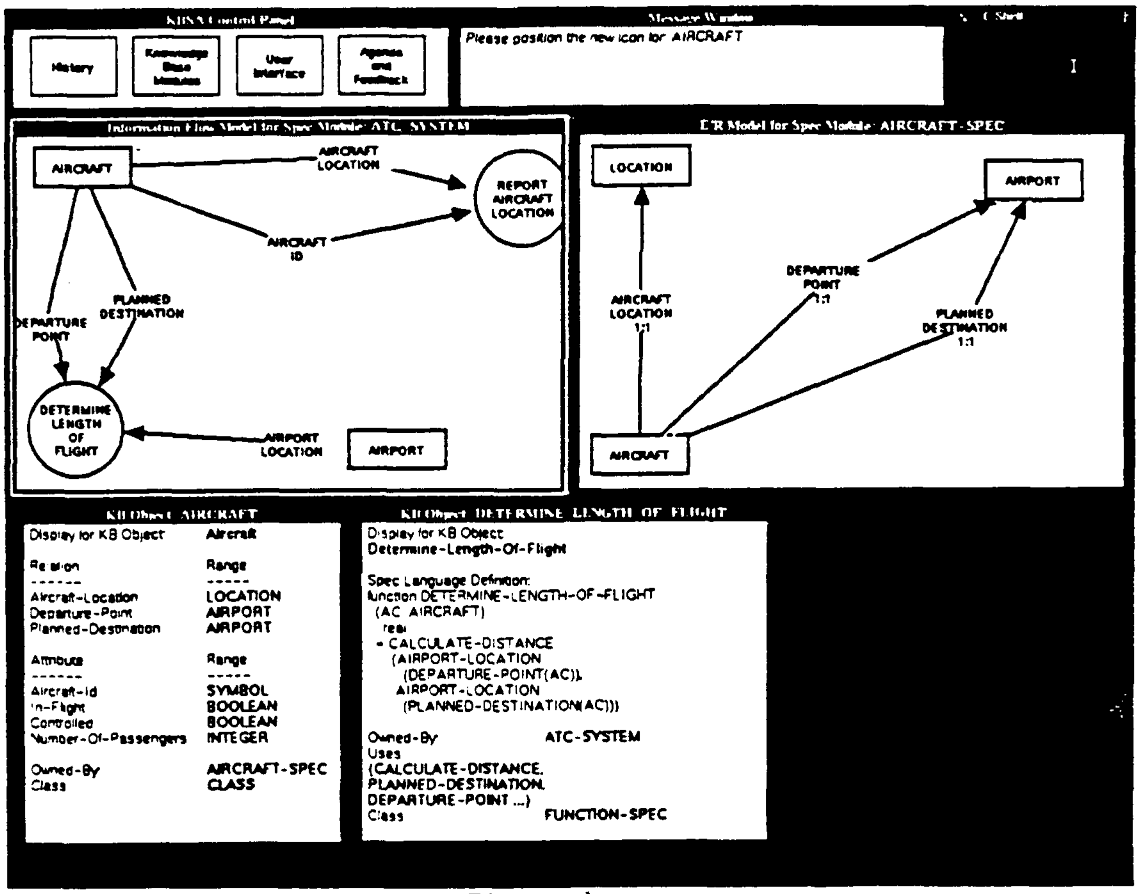
Next to the information flow diagram is an E/R diagram viewing some of the same classes and relations as the information flow diagram. The convention in our E/R diagrams is that an arrow is drawn pointing to the class that serves as the range of the relation and that cardinality is described with the name of the relation. E.g., AIRCRAFT LOCATION is a 1:1 relation with a domain of AIRCRAFT and range of LOCATION.

Beneath the information flow diagram is an object frame display for the AIRCRAFT class. Next to it is a syntax based display for DETERMINE LENGTH OF FLIGHT.

2.4.2 Results of the Bundle Evolution Transformation

Figure 2 shows the same interface after execution of the *Bundle* evolution transformation. Bundle takes a group of slots defined for a particular class (called the start-class) and changes the domain of the slots to be a new class (called the bundle-class). In addition, it creates a new relation (called the bundle-relation) which has the start-class as its domain and the bundle-class as its range. All of the process oriented specification objects which access or modify the bundled slots are transformed so that they use the bundle-relation to get from the start-class to the bundle-class.

In this example AIRCRAFT was the start-class; PLANNED DESTINATION, DEPARTURE POINT, and AIRCRAFT ID, were the bundled slots; FLIGHT PLAN was the bundle-class and AIRCRAFT FLIGHT PLAN was the bundle-relation. DETERMINE LENGTH OF FLIGHT is a process oriented specification object that was altered as a result of the transformation. E.g., the expression: DEPARTURE-POINT(AC) was transformed to: DEPARTURE-POINT(FLIGHT-

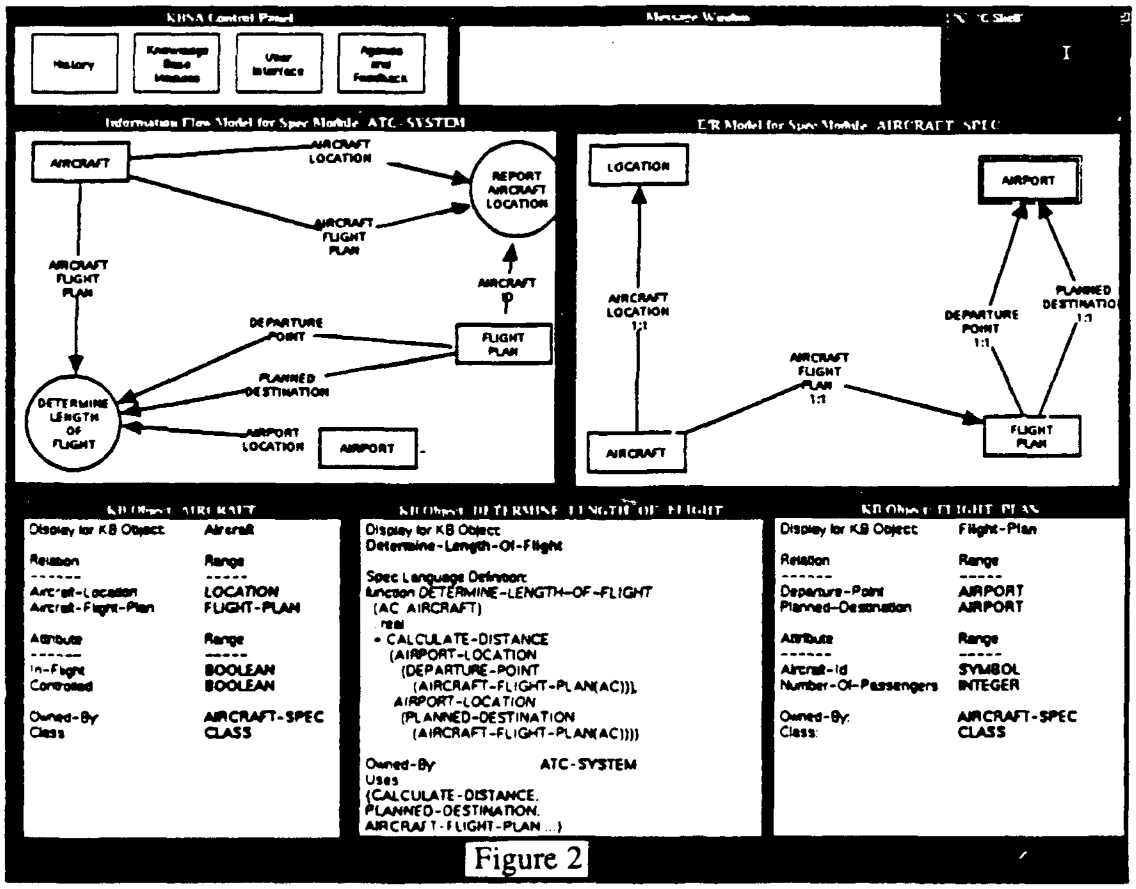


Please position the new icon for AIRCRAFT

```
Display for KB Object        Aircraft

Relation                     Range
------                       -----
Aircraft-Location            LOCATION
Departure-Point              AIRPORT
Planned-Destination          AIRPORT

Attribute                    Range
------                       -----
Aircraft-Id                  SYMBOL
In-Flight                    BOOLEAN
Controlled                   BOOLEAN
Number-Of-Passengers         INTEGER

Owned-By                     AIRCRAFT-SPEC
Class                        CLASS
```

```
Display for KB Object
Determine-Length-Of-Flight

Spec Language Definition:
function DETERMINE-LENGTH-OF-FLIGHT
  (AC AIRCRAFT)
   real
  = CALCULATE-DISTANCE
    (AIRPORT-LOCATION
      (DEPARTURE-POINT(AC)),
     AIRPORT-LOCATION
      (PLANNED-DESTINATION(AC)))

Owned-By                ATC-SYSTEM
Uses
(CALCULATE-DISTANCE,
PLANNED-DESTINATION,
DEPARTURE-POINT ...)
Class                   FUNCTION-SPEC
```

Figure 1



```
Display for KB Object        Aircraft

Relation                     Range
------                       -----
Aircraft-Location            LOCATION
Aircraft-Flight-Plan         FLIGHT-PLAN

Attribute                    Range
------                       -----
In-Flight                    BOOLEAN
Controlled                   BOOLEAN

Owned-By                     AIRCRAFT-SPEC
Class                        CLASS
```

```
Display for KB Object
Determine-Length-Of-Flight

Spec Language Definition:
function DETERMINE-LENGTH-OF-FLIGHT
  (AC AIRCRAFT)
   real
  = CALCULATE-DISTANCE
    (AIRPORT-LOCATION
      (DEPARTURE-POINT
        (AIRCRAFT-FLIGHT-PLAN(AC))),
     AIRPORT-LOCATION
      (PLANNED-DESTINATION
        (AIRCRAFT-FLIGHT-PLAN(AC))))

Owned-By                ATC-SYSTEM
Uses
(CALCULATE-DISTANCE,
PLANNED-DESTINATION,
AIRCRAFT-FLIGHT-PLAN ...)
```

```
Display for KB Object        Flight-Plan

Relation                     Range
------                       -----
Departure-Point              AIRPORT
Planned-Destination          AIRPORT

Attribute                    Range
------                       -----
Aircraft-Id                  SYMBOL
Number-Of-Passengers         INTEGER

Owned-By:                    AIRCRAFT-SPEC
Class:                       CLASS
```

Figure 2

PLAN(AC)).

As figure 2 shows, the presentation interface has updated all of the visible presentations so that they are consistent with the changes in the specification that resulted from the transformation.

2.4.3 Hypertext, Invariants, and Demons

Figure 3 shows an example of a hypertext node along with an invariant and a demon. Directly beneath the control panel is a simple hypertext requirement. Its hyperstring states that "All aircraft that are in flight must be controlled." The references to IN FLIGHT, AIRCRAFT, and CONTROLLED in the hyperstring are in bold face. This indicates reference to specification objects in the knowledge base. The developer could mouse on any of these references and get options to display the object using various presentations, to paraphrase it, to initiate evolution transformations, etc.

Beneath the informal hypertext node is a syntax display of the (formal) invariant that implements it. Next to the invariant is a demon created by invoking the Maintain Invariant Reactively[4] evolution transformation on it.

Next to the hypertext node is a semantic net graph showing the links between the hypertext requirement, the invariant that implements it and the demon that maintains the invariant. Note that these links can also be seen by looking at the displays for those objects.

3.0 Conclusion and Future Work

Our future work can be divided into three categories:

1) Extensions and enhancements to our existing requirements and specification functionality.

2) Integration of software from other KBSA related projects.

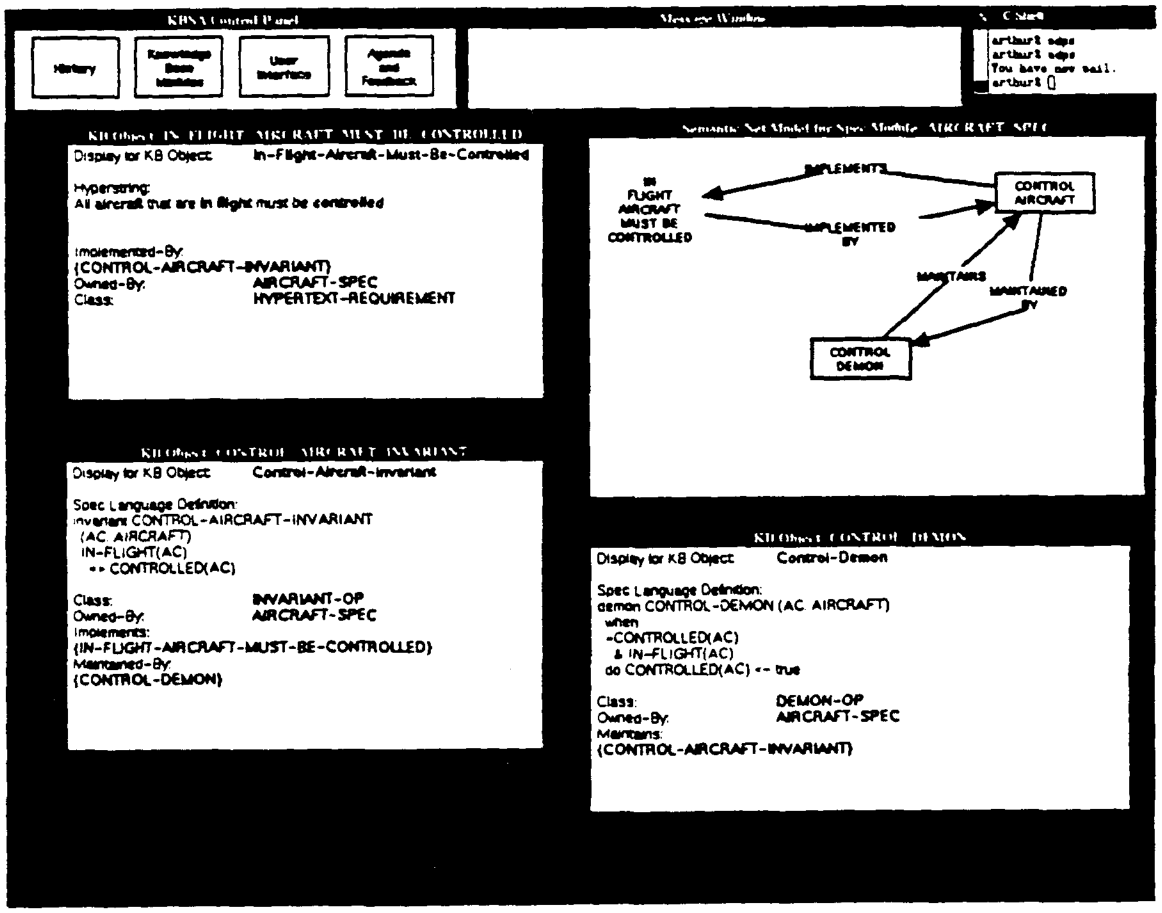
3) Research into important issues that have not been investigated in depth by previous KBSA projects, which we consider especially relevant to the Demonstration System goals described at the beginning of this paper.

3.1 Extensions to Requirements and Specification Functionality

The most important planned extensions to existing functionality are:

- ERSLa Language. As we described in section 2.2.1, we intend on

4. Described in more detail in [Johnson 88] and [Williams 90b].



```
arthur% sdps
arthur% sdps
You have new mail.
arthur% []
```

KBObject IN-FLIGHT-AIRCRAFT-MUST-BE-CONTROLLED

```
Display for KB Object       In-Flight-Aircraft-Must-Be-Controlled

Hyperstring:
All aircraft that are in flight must be controlled


Implemented-By:
{CONTROL-AIRCRAFT-INVARIANT}
Owned-By:                   AIRCRAFT-SPEC
Class:                      HYPERTEXT-REQUIREMENT
```

KBObject CONTROL-AIRCRAFT-INVARIANT

```
Display for KB Object       Control-Aircraft-Invariant

Spec Language Definition:
invariant CONTROL-AIRCRAFT-INVARIANT
 (AC: AIRCRAFT)
 IN-FLIGHT(AC)
  => CONTROLLED(AC)

Class:                      INVARIANT-OP
Owned-By:                   AIRCRAFT-SPEC
Implements:
{IN-FLIGHT-AIRCRAFT-MUST-BE-CONTROLLED}
Maintained-By:
{CONTROL-DEMON}
```

KBObject CONTROL-DEMON

```
Display for KB Object       Control-Demon

Spec Language Definition:
demon CONTROL-DEMON (AC: AIRCRAFT)
 when
 ~CONTROLLED(AC)
  & IN-FLIGHT(AC)
 do CONTROLLED(AC) <- true

Class:                      DEMON-OP
Owned-By:                   AIRCRAFT-SPEC
Maintains:
{CONTROL-AIRCRAFT-INVARIANT}
```

Figure 3

adding Gist-like constructs for representing time, and relations to ERSLa. In addition, we plan on utilizing the Refined-Gist representation in Aries [Johnson 90a] in order to develop automated conversion of Aries specifications to ERSLa. This will allow us to take advantage of the large reusable specification library being developed in Aries. We are also considering incorporating capabilities from classification based knowledge representation languages such as Loom [Macgregor 87] as the Aries project has done.

- Reusability. We will develop libraries of specifications and requirements, as well as aids for retrieving and adapting components from those libraries. In constructing tools for retrieving and adapting reusable components we will concentrate on two areas of research - Retrieval by Reformulation [Yen 88] and Case Based Reasoning [Riesbeck 89].

- Extended Scenarios. Our air traffic control scenarios will be improved. In addition, we will add scenarios in at least one other domain.

- Domain Specific Presentations. General purpose specification languages such as Gist and ERSLa are useful for professional developers. However, they are not useful for the typical computer user. For such users it is necessary to construct domain specific presentations (e.g., spread sheets, interface tool kits, circuit diagrams) that will allow end users to develop applications by interacting with user friendly presentations that resemble constructs and interfaces with which they are already familiar.

3.2 Integration of Software

3.2.1 Integration of PMA

We will integrate the final version of the PMA into the Demonstration System. This integration will include the following:

- Adapting the PMA interface to make it more consistent with conventions used in our interface and to add general interface capabilities developed in the Rapid Prototype.

- Adapting the facilities in the PMA so that we can represent our Process Model [Sasso 90] in the PMA.

- Enhancing our agenda mechanism so that its model of tasks, subtasks, agents, etc. is consistent with that of the PMA.

- Adapting PMA so that it is capable of placing tasks on the agenda mechanism and so that it can incorporate feedback from the agenda mechanism on completion of tasks.

3.2.2 Integration of KIDS

After the Rapid Prototype we will begin incorporation of parts of the Kestrel Interactive Development System (KIDS) in order to demonstrate the features of KBSA for transformation of high level specifications into optimized code. The biggest potential problem in this integration will be the clash between the functional programming paradigm used in KIDS and the state based paradigm used in Gist and ERSLa.

3.2.3 Integration of Other Software

We will consider the possible inclusion of other software, including software not based in Refine. Candidates include software developed in the Aries project for requirements organization and a theorem prover (many possible sources) to provide formal validation of specifications.

3.3 New Functionality

Two areas which deserve attention are providing guidance to the developer, and group development.

In order to provide guidance we will extend our Process Model [Sasso 90] and explicitly represent it using AI planning constructs such as states, goals, and operators. This should enable us to reason from the current state of development along with the developers goals (described by the developer and by PMA) to plans (sequences of evolution transformations) for achieving those goals.

As described in [Mui 89] KBSA research has paid little attention so far to issues related to programming in the large. One way to address this is to build tools for group development which utilize new ideas from *Groupware* interfaces [Ellis 88] into the Demonstration System. Two problems that we are considering are merging parallel development histories from different developers and development of a tool for group discussion and analysis of requirements (possibly incorporating ideas from Joint Application Design [Wood 89]).

3.4 Conclusion

This paper has described the functionality of the Concept Demonstration Rapid Prototype system. This system reimplements functionality developed in other KBSA projects dealing with requirements and specifications and will serve as the foundation for a Demonstration System that will exhibit the complete range of KBSA functionality.

Acknowledgements

Gerry Williams assisted in the development of the Rapid Prototype and with Jay Meyers converted the Aries Gist paraphraser to paraphrase ERSLa. Gordon Kotik provided invaluable assistance in describing how to develop evolution transformations in Refine and how to develop extensions to the Refine language. Beth Lange provided very useful comments on early versions of this paper and advice on various implementation problems that came up during development of the prototype. Lewis Johnson provided feedback on how to reimplement functionality from SpecA and Aries. Bill Sasso helped develop the hypertext mechanism. Chunka Mui provided valuable comments on previous versions of this paper. Xiaolei Qian developed extensions to the PMA constraint mechanism that made it possible for us to adapt it for use as the implementation behind the INVARIANT and DEMON ERSLa constructs. Marilyn Daum and Richard Jullig helped us integrate the PMA.

References

Blaine L. , Goldberg et al., "Progress on the KBSA Performance Estimation Assistant", Proceedings of the 3rd Knowledge-Based Software Assistant Conference, 1988.

Cohen, D. *AP5 Manual,* USC Information Sciences Institute, 1985.

Conklin, J., and M. Begeman. "gIBIS: A Hypertext Tool for Exploratory Policy Discussion." *Proceedings of the Conference on Computer Support for Cooperative Work.* Portland, OR: 1988, pp. 140-152.

Ellis, Clarence, Gibbs, Simon, Rein, Gail, "Groupware: The Research and Development Issues", MCC Technical Report STP-414-88, December 1988.

Fikes, Richard, Kehler, Tom, "The Role of Frame-Based Representation in Reasoning", Communications of the ACM, September 1985, Volume 28, Number 9.

Gilham, Li-mei, Jullig, Richard, Ladkin, Peter, Polak, Wolfgang, "Knowledge-Based Software Project Management", Kestrel Institute Technical Report: KES.U.87.3, November, 1986.

Gilham, Li-mei, Goldberg, Allen, Wang, T.C., "Toward Reliable Reactive Systems", Proceedings of the 5th International Workshop on Software Specifications and Design, Pittsburgh, PA., May 1989.

Harris, D. & Runkel, J., "An Introduction to the Knowledge Based Requirements Assistant Capabilities", Rome Air Development Center, Contract No. F30602-85-C-0267, September, 1988.

Johnson et al., "The Knowledge-Based Specification Assistant, Final Report", Rome Air

Development Center, Contract No. F30602-85-C-0221, 1988.

Johnson, Lewis, and Harris, Dave, "Requirements Analysis Using Aries: Themes and Examples", Proceedings of the 5th Annual Knowledge Based Software Assistant Conference. September, 1990(a).

Johnson, Lewis, and Feather, Martin, "Using Evolution Transformations to Construct Specifications", in *Automating Software Design*, Edited by Michael Lowry and Robert McCartny, AAAI Press, 1990(b).

MacGregor, R, Bates, R., 1987. "The Loom Knowledge Representation Language", In *Proceedings of the Knowledge-Based Systems Workshop*, St. Louis, Missouri, April 21-23, 1987, ISI/RS-87-188.

Mui, C. and M. DeBellis. "KBSA Technology Transfer: An Industry Perspective", Proceedings of the 4th Annual Knowledge Based Software Assistant Conference. September, 1989.

Reasoning Systems Inc., "Refine User's Guide", June 15, 1986.

Riesbeck, Chris and Schank, Roger, *Inside Case Based Reasoning*, Lawrence Erlbaum, New Jersey 1989.

Sasso, W. and M. DeBellis. A Software Development Process Model for the KBSA Concept Demonstration System. Proceedings of the 5th Annual Knowledge Based Software Assistant Conference. September, 1990.

Smith Doug, "Automating the Development of Software", Proceedings of the 5th Annual Knowledge Based Software Assistant Conference. September, 1990.

Williams, Gerald. and Myers, Jay, J, Exploiting Language Metamodel Correspondences to Provide Paraphrasing Capabilities for the Concept Demonstration Project. Proceedings of the 5th Annual Knowledge Based Software Assistant Conference. September, 1990a.

Williams, Gerald, "KBSA Concept Demonstration System Manual - Draft Version", Andersen Consulting CSTaR Technical Report, 1990b.

Wood, Janet and Denise Silber, "Joint Application Design: How to Design Quality Systems in 40% Less Time", John Wiley and Sons, 1989, New York, NY.

Yen, J., Neches, R., DeBellis, M., "Specification by Reformulation: A Paradigm for Building Integrated User Support Environments", Proceeding of the 7th National Conference on Artificial Intelligence, Saint Paul, Minnesota, Aug 1988.

A Software Development Process Model for the KBSA Concept Demonstration System[1]

William C. Sasso and Michael DeBellis

Center for Strategic Technology Research
Andersen Consulting
100 South Wacker Drive, 9th Floor
Chicago, IL 60606

1. Introduction

This paper presents our preliminary vision of a process model for software development using a Knowledge Based Software Assistant (KBSA). Further, it relates that process model to the functionality present in the KBSA Concept Demonstration Rapid Prototype and the additional functionality we plan to include in the deliverable Concept Demonstration System.

This paper does not attempt to present the definitive KBSA software development process model, but rather a convincingly detailed example of one possible model. As other KBSA researchers have noted, there will probably never be a single KBSA process model – one of KBSA's greatest strengths is its potential ability to support alternative process models (Jullig, 1989). However, in order to understand and discuss the issues involved in developing process models for KBSA, we believe it is essential to develop at least one detailed example of a potential model.

Distinctions Between the KBSA and Waterfall Models. The KBSA-oriented software development process model discussed below presents its users with three major advantages relative to the conventional Waterfall development practices:

- Formal representation and manipulation of the target system specification without committing to a specific implementation, enabling strong validation of via simulation, prototyping, and other evaluation techniques and system-level (rather than program-level) optimization for better performance;

- Tightly integrated, rapidly cyclical, incremental development, enhancing continuity between successive development artifact states and enabling improved traceability and replay of the development process; and

- Sophisticated and integrated process modeling capabilities, allowing control of development without forcing a waterfall-like, lock-step progression on the set of development artifacts.

1. The work reported in this paper was supported by RADC contract F30602-89-C-0160.

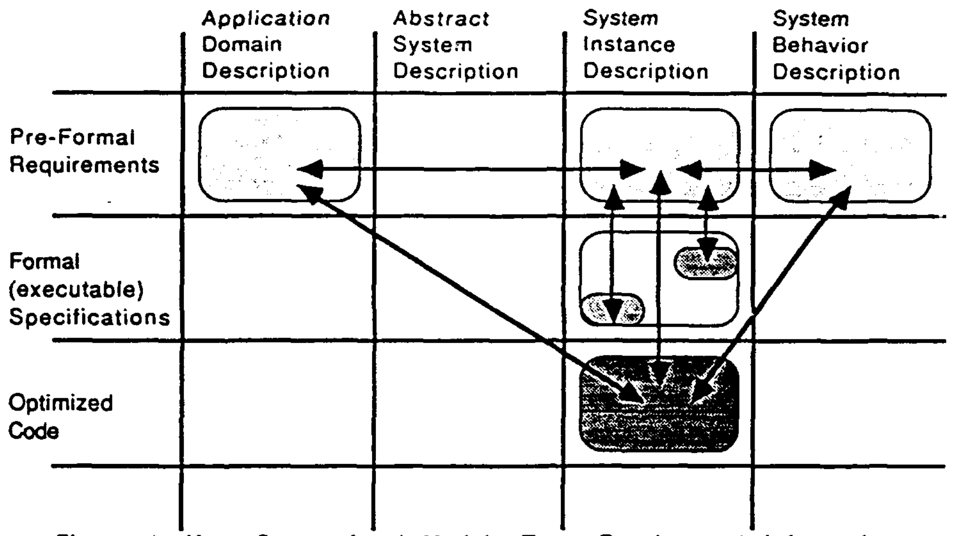

Application Domain Description
Abstract System Description
System Instance Description
System Behavior Description
Pre-Formal Requirements
Formal (executable) Specifications
Optimized Code

Figure 1: How Conventional Models Treat Development Information

As shown in Figure 1, conventional, waterfall development typically captures three types of information defining the system to be built: an application domain model (e.g., an entity-relation diagram), a system structure model (e.g., a Yourdan structure chart), and a system behavior model (e.g., a functional specification). This information is captured, organized, and integrated at an informal level, and the system structure model is then elaborated into optimized code. In some cases, parts of the system structure model are elaborated into formal specifications via prototyping or simulation models. These are (sometimes) used to validate the informal system structure model, prior to its implementation and optimization in code.

Figure 2 shows that KBSA handles this information in a significantly different way. First, it makes far greater use of the formal specification level. This enables incremental development and increases the power of the validation process. Second, KBSA makes explicit use of a new category of information, the abstract system model. The abstract system model is an formal specification, synthesizing the desired system behavior and the key application domain models, prior to any major commitment to implementation techniques. This enables the KBSA-supported developer to defer optimization issues until the abstract system model has been elaborated and validated.

Structure of the Paper. The next section describes briefly the previous work from which our model is derived, and presents our process model at an overview level, depicting the complete spectrum of software development and evolution. We then present more detailed discussions of the model's treatment of requirements and implementation-independent specifications. Within the Concept Demonstration project, these processes have received the bulk of our effort and attention to date, and are thus more fully described than is the next one, specification implementation, which will become a major area of effort for our project in the coming year. Finally, we discuss how this model addresses issues associated with programming in the large, since its support is a major objective of KBSA.

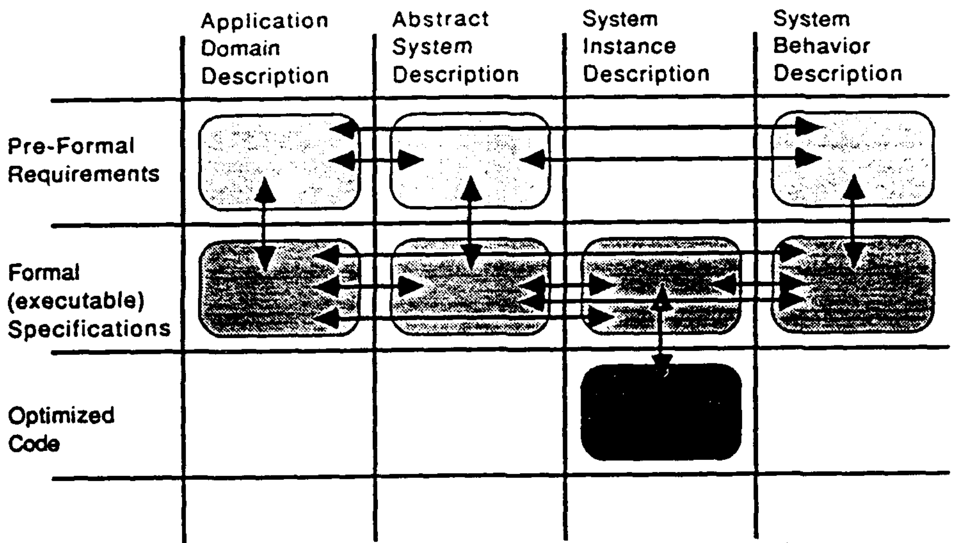

Application Domain Description
Abstract System Description
System Instance Description
System Behavior Description
Pre-Formal Requirements
Formal (executable) Specifications
Optimized Code

Figure 2: How KBSA Treats Development Information

2. Presentation of the Process Model

This process model has several important antecedents. Balzer (1985) presents a preliminary model of software development based on formal methods, and extended it by recognizing the importance of specification validation, the advantages of maintenance at the specification level, and the idea of capturing and replaying sequences of formal development activities. This extended model is presented in Figure 3. The approach embodied in the earlier Knowledge-Based Software Assistant (Green et al, 1983) program vision maps into Balzer's model easily. Similarly, Zave's (1984) discussion of operational specifications as a basis for software development superior to conventional approaches meshes well with his model.

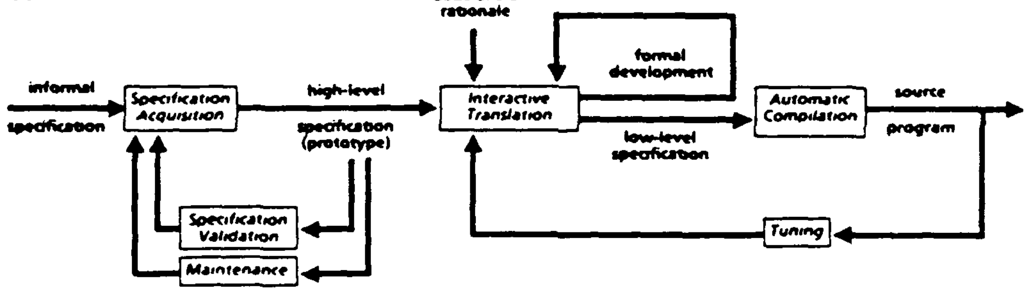


Figure 3: Balzer's Extended Automatic Programming Paradigm

This model is a direct descendent of Balzer's. It extends his work by explicitly considering (1) the acquisition and organization of informally expressed requirements, (2) the communication and contention issues of programming in the large, (3) the reuse of specification elements, and (4) the inclusion of reengineering as a means of developing formal specifications. The process model we present here is discussed primarily in terms of the forward engineering process, but it has strong assumptions concerning the importance of reverse engineering and the evolution of deployed systems.

Overview of a KBSA-Oriented Process Model. Figure 4 represents a high-level view of the development process. The basic concept illustrated in the Figure is that KBSA-based development progresses through four major activities:

1. Acquisition: capture of requirements, in text and/or graphic form, followed by their disambiguation and organization into a structure of well-formed requirements clusters, as a preparatory step towards formalization;

2. Specification development: formalization of requirements into specifications and their incremental development via evolution transformations (cf. Johnson and Harris, 1990) to a detailed but still implementation-independent state;

3. Specification implementation: optimization of those detailed specifications via meaning-preserving transformations into an implementation-specific state suitable for compilation and operational deployment; and

4. Deployment and installation of the production-quality system, followed by its ongoing operation.

An organizational process change activity is shown in the Figure, reflecting recognition that new application software will generally imply adjustments to the organization's operating procedures, training programs, hiring practices, and possibly other areas as well. In addition, plans for conversion and installation should be developed prior to the deployment step. We will not address this activity further in this paper, but believe that it will require additional consideration before KBSA technology can be successfully transferred to an industrial context.

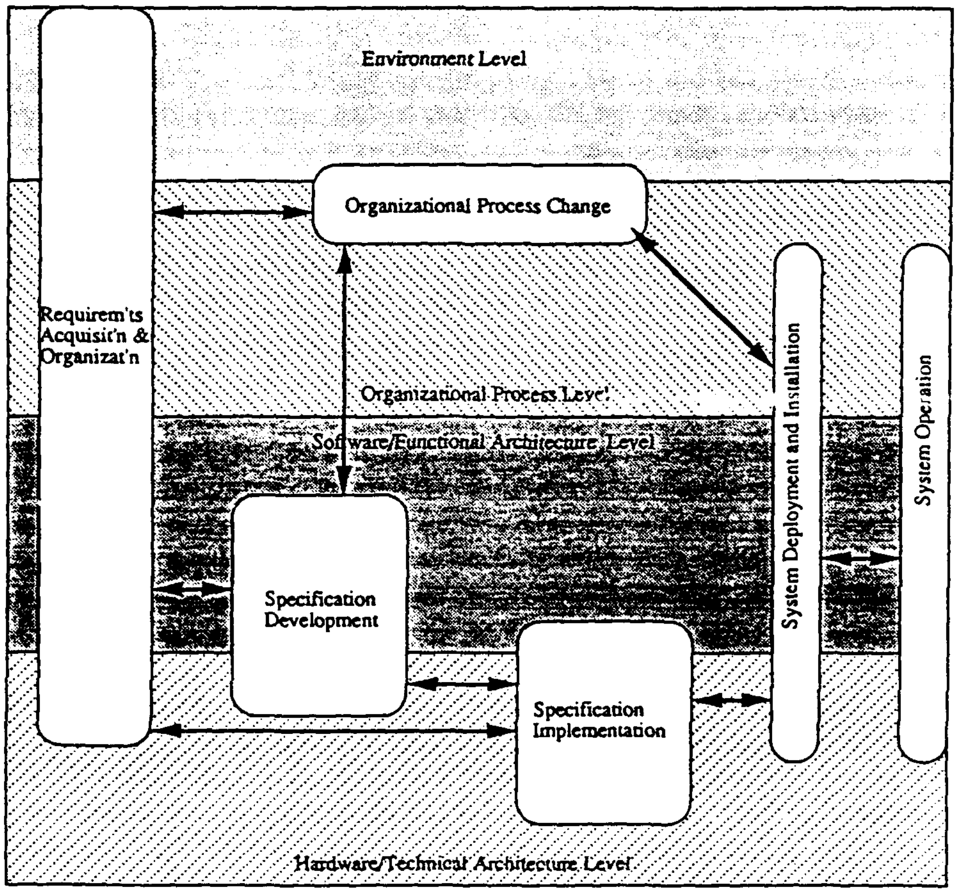
Reverse engineering is the process of taking existing systems, abstracting their designs from existing code and documentation, and reimplementing them in a form that is easier to maintain and enhance. Reverse engineering will be essential to allow organizations to transition from software developed using traditional techniques to a KBSA approach. Most organizations have very large amounts of existing software that is crucial to their operation but exists in out-dated environments that are very difficult to maintain. For such organizations to use KBSA only for completely new systems will cause them to miss most of the benefits of KBSA. This is the case because:

1) Up to 80% of the effort spent on software development is spent on maintaining existing software as opposed to implementing new software (Martin and McClure 1983).

2) Even completely new systems developed with KBSA will have to integrate with existing systems. If these existing systems remain in out-dated non-KBSA technologies, such integration will be far more difficult and will miss benefits of KBSA such as a domain model shared by multiple applications.

While the importance of reverse engineering been articulated by KBSA related researchers (Kozaczynski and Ning, 1989; Kotik and Markosian, 1989; Newcomb, 1989), much work remains to be done to develop capabilities for reverse engineering into a KBSA environment.

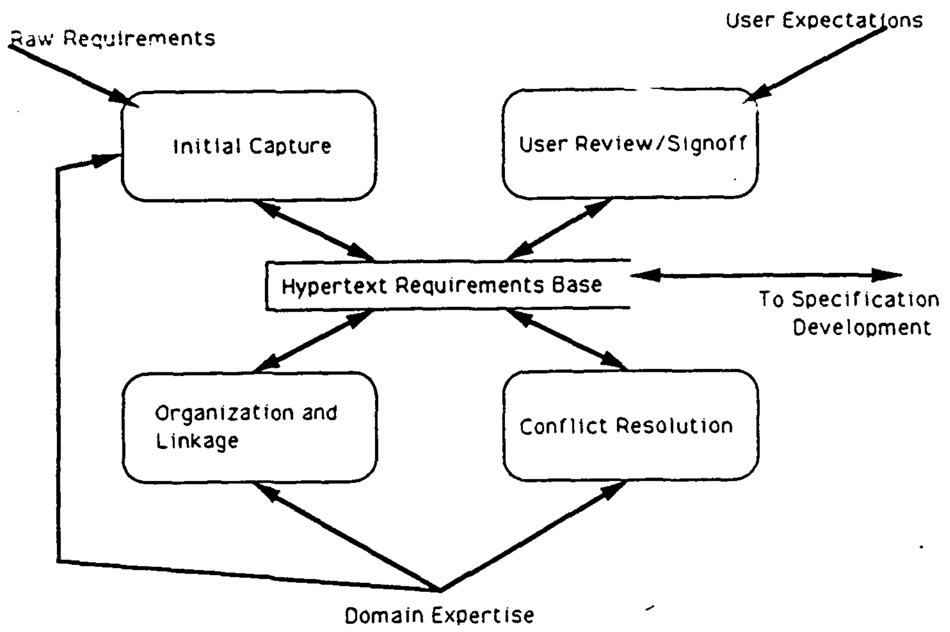
Figure 4: Process Model Overview



Environment Level
Organizational Process Change
Requirem'ts Acquisit'n & Organizat'n
Organizational Process Level
Software/Functional Architecture Level
System Deployment and Installation
System Operation
Specification Development
Specification Implementation
Hardware/Technical Architecture Level

3. Requirements Acquisition and Organization

Requirements acquisition and organization is the first step in the formal development process. As indicated in Figure 5, it involves the initial identification and organization of requirements, such as interview and observation data and application domain descriptions in natural language. Requirements can be differentiated into three types of information: application domain models, user expectations of system behavior, and the designer's preliminary concept of the evolving system structure (Johnson and Harris, 1990, Johnson and Feather, 1990). Each requirement can be linked with others to form clusters of related requirements. These can then be organized into multiple, overlapping sets of related requirements for use by different development stakeholders. Below is a set of activities which might be used to capture, organize, and refine a typical requirement. This is not a fixed sequence of activities to be applied mechanically to every requirement, but rather a set of operations to be applied in various orders as the nature of the case requires. Figure 5: Requirements Acquisition and Organization



- Initial capture of an atomic requirement element. For example, "The minimum vertical aircraft separation should be 1,000 ft up to flight level 290." We would also capture information describing the source or owner of the requirement (in this case, FAA document 7110.65, page 5-5-1).

- Objective (automated) clustering by ownership and commonly occurring terms. For example, the sentence above might be automatically clustered with other declarations concerning "minimum ... aircraft separation."

- Subjective (assisted) clustering on the basis of conceptual relatedness. For example, we might link aircraft separation rules with a domain model of the jet-route network, since the capacity of the jet-routes can only be determined with reference to the minimum separation standards in effect.

- Identification and resolution of incompleteness and incorrectness. In terms of the above example, we might note that flight levels are units of altitude measurement corresponding to 100 feet (i.e., flight level 290 is 29,000 feet).

- Resolution of conflicts/contradictions internal to clusters. Suppose, for example, that one requirement states that "All indicators of aircraft, airspace, and radar will appear on the controller's display at all times" while another states that "To improve usability, no more than 50 distinct items will be displayed at any given time." These are potentially in conflict, should the airspace region displayed contain more than 50 items. We might resolve this by requiring that the system allow the "graying out" of some of the less important items when the display includes more than 50 items, or by providing several types of display filtering options.

- Review and approval of individual requirements clusters. For instance, after the appropriate stakeholders have signed off on the separation requirements, they can begin the specification development activity.

- Resolution of conflicts/contradictions between clusters. For example, the separation cluster and the human-computer interaction standards cluster might contain contradictory expectations regarding the interface design. These will need to be addressed (at least to the extent of identifying tradeoffs between different goals of the target system) before the modeling and evaluation of the conflict in the specification development process can take place.

Review and signoff on a specific requirement cluster will indicate that it is ready to enter the specification development process, and estimate its centrality or importance to the overall development effort. At this point, it will be placed in a modification controlled state. Certain automated analyses will be conducted in an ongoing fashion to ensure, for example, that each requirements cluster is well-formed.

The KBSA Concept Demonstration System will support requirements acquisition and organization via a hyperstring capability (Johnson et al, 1990) and the use of a presentation architecture (Harris and Czuchry, 1988). A general requirements browsing capability will be available at all times.

4. Specification Development

Once a cluster of requirements has been approved by its stakeholders, the process of formal specification can begin. As shown in Figure 6, the specification development activity provides two main approaches to development of the formal specification: reuse and construction. After one (or possibly both) of these approaches has been selected, the formal specification is developed and extended. Periodically, it will be validated by the owners of its associated requirements. Typically, the specification will cycle through several validation and revision processes, as inconsistencies are uncovered and corrected. Once the cluster has been transformed into an accepted formal spec module, it will be combined with other modules in specification integration process designed to remove any inconsistencies. Finally, any remaining elements of incomplete specification (such as unintentional non-determinism, omniscience, and omnipotence) should be corrected in a specification completion task (Johnson, 1990).

Non-determinism occurs, for example, when a specification defines a set of distinct options without indicating how selection among them is to be made (e.g., a constraint stating that there exists some controller for each controlled aircraft, without stating how a controller is to be assigned to an aircraft). An omniscient specification assumes that it has access to any data it requires, ignoring the question "Where did that data come from?" An omnipotent specification is one which assumes that a change made in its knowledge base (e.g., a change in the value of aircraft-heading) will automatically cause the corresponding change in the environment (the physical aircraft will actually change its heading).

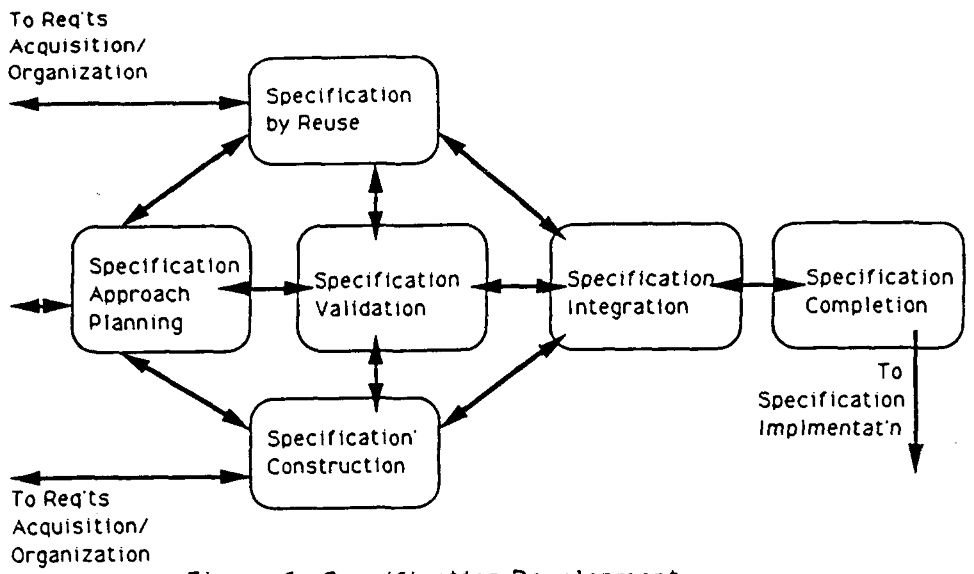

To Req'ts Acquisition/ Organization
Specification by Reuse
Specification Approach Planning
Specification Validation
Specification Integration
Specification Completion
To Specification Implmentat'n
Specification Construction
To Req'ts Acquisition/ Organization

Figure 6 : Specification Development

A typical specification element will begin as an informally expressed requirement. Via the reuse approach, it will be developed via the following operations:

- Identify parameter values for search and retrieval of reusable elements.
- Evaluate alternative elements for reuse and select most appropriate one(s).
- Adapt to current context by replacement of generic and default terms with terms specific to the particular application domain, as a side-effect linking reused elements to the requirements they specify.

Using the incremental construction approach, elements of a requirements cluster will be formalized, extended, and validated via a combination of the following operations:

- Use basic Knowledge Base manipulation commands to create formal elements corresponding to the objects and processes in the requirement.
- Use evolution transformations to elaborate these elements, replace their default values with appropriate context-specific ones, and capture their complex relationships (e.g., define invariants).

In terms of elaborating, validating, organizing, and completing the formal specification, a common set of activities will be performed, whether the first-cut formal spec was developed via reuse, construction, or (most likely) some combination of these:

- Group the formal elements into spec modules corresponding to a cluster.
- Ensure completeness and consistency of each formal spec module.

- Present for validation by the stakeholders of the requirements cluster via paraphrase, prototype, and/or simulation.

- Revise as necessary, in some cases adding or modifying elements in the requirements cluster, until all stakeholders approve the specification module relative to its requirements cluster.

- Integrate the individual specification modules into a "spec complex," removing any inconsistencies between the modules, and validating that the behavior of the spec complex continues to satisfy the associated requirements clusters.

- Identify any remaining non-determinism or elements with omnipotence or omniscience, and replace these elements as appropriate.

In the KBSA Concept Demonstration System, basic knowledge-base manipulation commands will be used to create the initial formal objects. These will be extended and refined via evolution transformations similar to the high-level editing commands developed in the Knowledge-Based Specification Assistant (Johnson, 1987) and the ARIES project (Johnson and Feather, 1989). We expect to use a retrieval-by-reformulation approach (Williams, 1984) to assist in the location of appropriate reusable elements, and are evaluating case-based reasoning (citation) as another technology with potential to support this task. Powerful validation techniques such as paraphrasing, prototyping, and simulation will complement the consistency and completeness analyses performed automatically by the KBSA Concept Demonstration System.

5. Specification Implementation

The formal specification passed forward to specification implementation will include very high level constructs, such as time constructs for historical reference, demons, constraints, and class hierarchies with inheritance and default values. Each of these constructs will have a default method of compilation. Some of these methods may be very slow, and none will be optimal. Specification implementation will transform these constructs into more efficient, lower level constructs via facilities such as data structure selection, unfolding of constraints, state saving (to handle historical reference), finite differencing, and loop fusion.

These techniques have been examined in the research performed at the Kestrel Institute in the KIDS environment (i.e., Goldberg, 1989) as well as at other research centers (cf. Balzer, 1985). The amount of effort required in this stage will depend upon a number of factors:

- Complexity of the problem. For example, implementation of a specification for a real time scheduling system will require all possible optimization techniques.

- Capabilities of the run-time environment. For example, in some environments it will not be possible to directly compile a specification with global knowledge base or constraint constructs.

- Non-functional requirements. For instance, response time, reliability, and other non-functional constraints will feed forward as parameters from the requirements acquisition and organization task.

- Sophistication of the compiler. Some of the transformations which are now user guided (such as those demonstrated in the KIDS environment) may eventually be automated, as compiler technology continues to become more powerful.

In contrast to the transformations used in specification development, those used here will be meaning preserving – ideally, this will have been proven to be true for them. They will seek to improve performance and other run-time characteristics of the target system. Most of the previous research on these transformations has emphasized processing optimizations, which are very important, especially for real-time systems. But other types of implementation transformations will also require attention before KBSA will be usable in industrial environments. These include data-oriented optimizations and system architecture selections. Systems which deal with very large amounts of data require different types of optimizations than do process intensive systems. These optimizations must consider low-level details about the structure and access languages of the system's various data-bases. System architecture factors play an essential part in the implementation decisions which real programmers make. KBSA needs to represent and use such knowledge to integrate different architectural environments efficiently and seamlessly.

6. Support for Programming in the Large

A KBSA-oriented process model should also provide explicit notions of support for programming in the large, i.e., for the situation where large teams of software developers work in parallel and interdependent fashion on the construction of large systems. Our process model includes several features which address this issue.

Synthesis of Parallel Development: In large projects, developers may need to work on the same specification elements simultaneously. At some later point, the changes they have made must be reconciled to maintain consistency of the specification.

Varied Phasing of Software Development: The conventional waterfall model assumes that software development proceeds in lockstep – that requirements for all elements of the system are completed before any of the design activities begin. This is believed necessary to ensure the conceptual integrity of the design, but is rarely enforced in practice. In the KBSA process model, the more powerful automated consistency maintenance capabilities available make it possible for us to allow different elements of the specification to progress at varied rates. For example, in a conventional CASE environment, there is no automated capability that can maintain consistency between pseudo-code and executable code – KBSA provides such a capability. Figure 7 depicts this. The layers of the pyramid represent the activities of software development and each shaded vertical line represents the state of progress of some specification element.

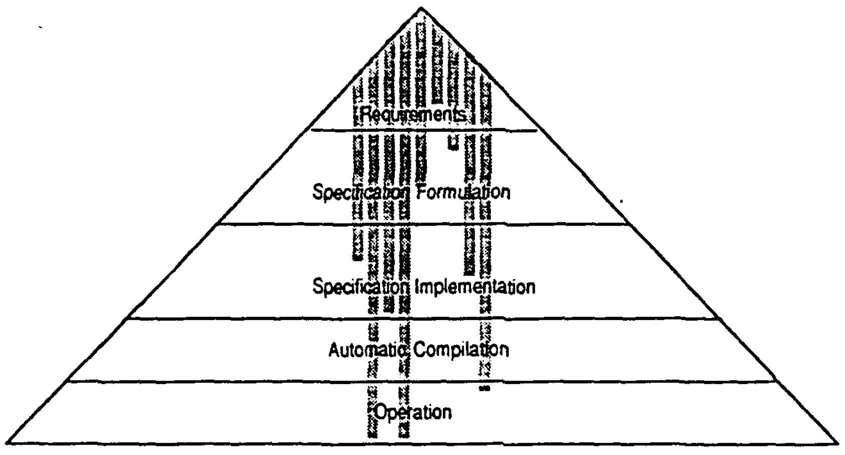


Figure 7: Elements at Different Stages of the Process Model

Integration of Project Management and Software Development: The varied phasing mentioned above facilitates the software developer's job, but complicates that of the project manager. Because KBSA maintains the development knowledge and the project management knowledge in an integrated knowledge base, more sophisticated project reporting and analysis is available (Jullig et al, 1987). Via the creation and automated maintenance of project models, project management can define, *monitor*, and *control* the project even under complex varied phasing.

6. Summary

This paper has presented a high-level overview of one possible process model for software development in a KBSA environment. The model's major activities are organized around the degree of formalization and elaboration of the software development artifact. They need not proceed in lockstep, but rather provide the developer with strongly integrated facilities enabling rapid cycles of software development, validation, elaboration, and ongoing enhancement. The model recognizes that support is needed for the acquisition and organization of informal requirements information as well as for the creation and extension of formally expressed specifications, and extends an antecedent model to do so. It recognizes the importance of interfacing with existing systems and the recovery of procedural knowledge embedded within them, and provides these capabilities by inclusion of Reverse Engineering as a first-level activity.

7. Acknowledgments

We gratefully acknowledge the contributions of Lewis Johnson, Bruce Johnson, Richard Jullig, and Gerry Williams to our understanding of the concepts presented in this paper.

Balzer, R. A 15 Year Perspective on Automatic Programming. IEEE Transactions on Software Engineering. SE-11(11): November 1985.

Feather, M. Detecting Interference When Merging Specification Evolutions. Proceedings, 5th International Workshop on Software Specification and Design. 1989.

Goldberg, A.. Reusing Software Developments. Proceedings of the 4th Annual Knowledge Based Software Assistant Conference. September, 1989.

Goldman, N. Three Dimensions of Design Development. ISI/RS-83-2. July, 1983.

Green, C. D. Luckham, R. Balzer, T. Cheatham, and C. Rich. Report on a Knowledge-Based Software Assistant. RADC-TR-83-195. August, 1983.

Harris, D. and R. Czuchry. Knowledge-Based Requirements Assistant. RADC-TR-88-205. October, 1988.

Johnson, W. Personal Communication on Partial Specification. August, 1990.

Johnson, W. Overview of the Knowledge Based Specification Assistant. Proceedings of the 2nd KBSA Conference. August, 1987.

Johnson, W., and M. Feather. Building Evolution Transformation Libraries. Proceedings of the 4th Annual Knowledge Based Software Assistant Conference. September, 1989.

Johnson, W., and M. Feather. Using Evolution Transformations to Construct Specifications. in Lowry, M. and R. Mccartney, eds. Automating Software Design. AAAI Press: 1990.

Johnson, W. and D. Harris. Requirements Analysis using ARIES: Themes and Examples. Proceedings of the 5th Annual Knowledge Based Software Assistant Conference. September, 1990.

Jullig, R. Progress on the Project Management Assistant. Proceedings of the 4th Annual Knowledge Based Software Assistant Conference. September, 1989.

Jullig, R., et al. KBSA Project Management Assistant. RADC-TR-87-78. July, 1987.

Kotik, G. and L. Markosian. KBSA for Automated Software Analysis, Test Generation, and Management. Proceedings of the 4th Annual Knowledge Based Software Assistant Conference. September, 1989.

Kozaczynski, W., and J. Ning. SRE: A Knowledge-Based Environment for Large-Scale Software Re-engineering Activities. Proceedings of the 11th International Conference on Software Engineering. May, 1989.

Martin, J. and C McClure, Software Maintenance: The Problem and its Solution. Prentice-Hall. Englewood Cliffs, NJ: 1983.

Mui, C., and M. DeBellis. KBSA Technology Transfer: An Industry Perspective. Proceedings of the 4th Annual Knowledge Based Software Assistant Conference. September, 1989.

Newcomb, P. Knowledge-Based Reverse Engineering for Re-engineering and Reuse. Proceedings of the 4th Annual Knowledge Based Software Assistant Conference. September, 1989.

Williams, M. What makes RABBIT run? International Journal of Man-Machine Studies. 21:333-352. 1984.

Zave, P. The Operational Versus the Conventional Approach to Software Development. Communications of the ACM. 27(2): February, 1984.

REVERSE SOFTWARE ENGINEERING OF CONCURRENT PROGRAMS

Dr. Xiang C. Ge. and Dr. Noah S. Prywes
Computer Command and Control Company
2300 Chestnut Street, Ste. 230
Philadelphia, PA 19103

Abstract

This paper describes developing a system for reverse software engineering of real time programs. It is addressed specifically to the US Navy's modernization of tactical and strategic systems. However, the problem is widely encountered. The Navy has a 30 year investment in knowledge and experience of having developed and operated these systems. The primary language used has been CMS-2. Modernization is required urgently for utilizing a more advanced multiprocessor distributed computer architecture programmed in Ada.

The approach here consists of completely automatic translation from CMS-2 into a user-oriented non-procedural specification language called MODEL. The user can then better understand, maintain and modernize the specification of programs in the MODEL language. There is an existing system for analysis, translation, and optimization in conversion of MODEL to Ada. This completes the automation from the real time CMS-2 programs to Ada programs.

However, mere translation does not suffice. The modernization staff must also have automation of testing, maintenance, documentation and training. This paper discusses first the integration of the translators of CMS-2 to MODEL and MODEL to Ada with Computer Aided Software Engineering (CASE) systems. The overall system will use a powerful workstation with graphics. This is followed by description of the analysis of the specifications of real-time programs for their understanding and validation.

1. Introduction

There is great interest in reverse software engineering and there is an enormous potential market for automating the modernization of software. This is a topic of research at a number of organizations. The approach is common: to develop an intermediate user oriented language and two translators, from the old software to the intermediate language, and from the intermediate language to the modernized software. The intermediate language used in this paper is called MODEL[3]. There is also a translator from MODEL to modernized software in Ada, C, or PL/1 for distributed computer architectures used in real-time applications[6]. A translation from CMS-2 to MODEL is being developed[15].

The paper is addressed specifically to the US Navy, but the approach is general and universal.

Over the next decade the US Navy will have to modernize its tactical and strategic computer systems. These systems represent an accumulation of 30 years development and experience and amount to 10-20 million program lines in CMS-2, using specialized ship-board computers. The Ada language has been selected for the next generation software development. The hardware architecture has yet to be selected. It will most likely involve a gradual transition from the present hardware to a new computer architecture that will incorporate much greater parallelism and computational power. The paper describes the automation of program translation from the CMS-2 language to the Ada language. The immense investment and accumulated knowledge of tactical and strategic systems can thus be captured and reused in modernized systems. This technology will greatly reduce the cost and accelerate the modernization of Navy tactical and strategic computerized systems.

Mere automatic translation from CMS-2 to Ada would not suffice. The automatic translator must also provide the staff that is performing the conversion with at least support for testing and maintenance of the newly generated programs. The testing is difficult due to critical timing in real-time systems and due to high reliability and confidence requirements. This will necessitate generating documentation of program functions and algorithms, generating test data and facilitating making modifications. Eventually, support must also be provided for modifying software architecture to optimize operation with the new hardware architecture. Section 2 describes an integrated system for automating the conversion.

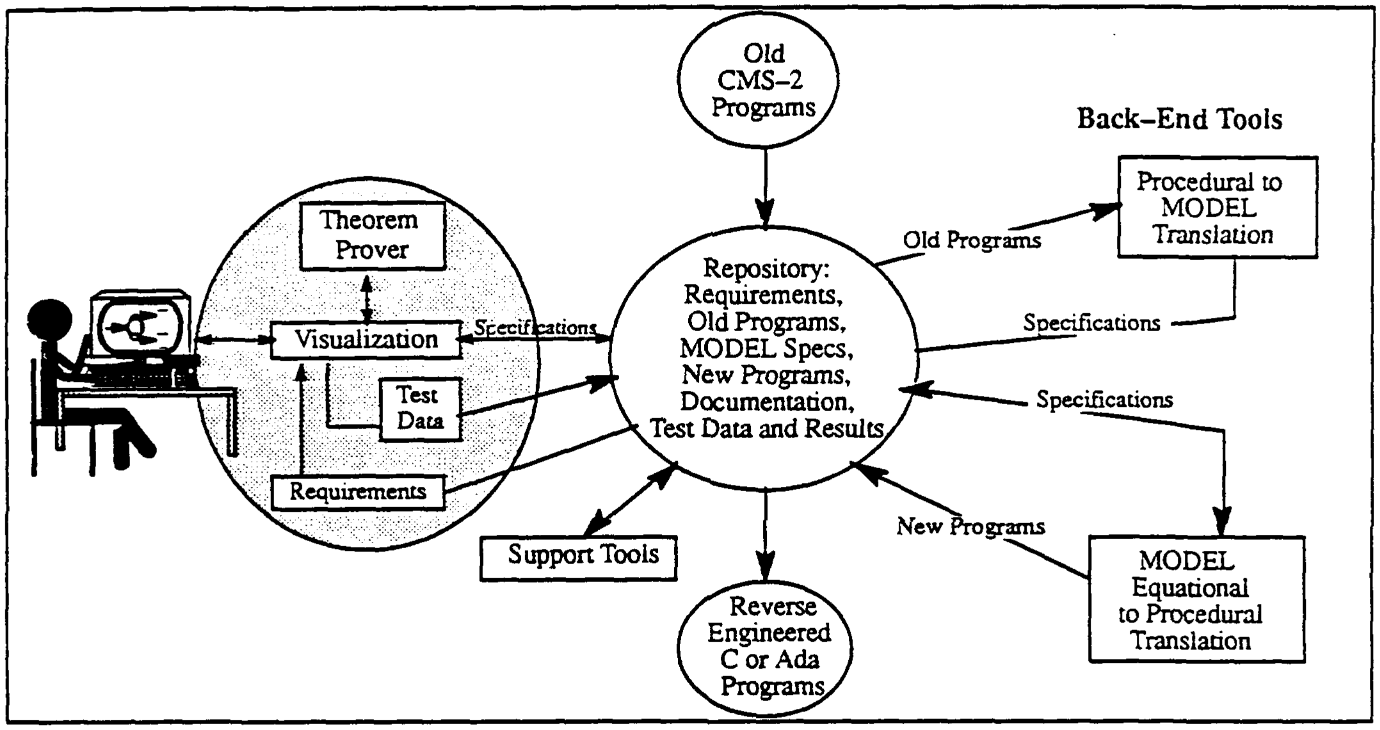

Old CMS-2 Programs
Back-End Tools
Procedural to MODEL Translation
Old Programs
Theorem Prover
Repository: Requirements, Old Programs, MODEL Specs, New Programs, Documentation, Test Data and Results
Specifications
Visualization
Specifications
Test Data
Specifications
Requirements
New Programs
Support Tools
MODEL Equational to Procedural Translation
Reverse Engineered C or Ada Programs

Figure 1: Illustration Of CMS–2 To Ada Reverse Engineering Prototype.

Figure 1 shows schematically the components of such a system. The flow of information is as follows. The CMS-2 programs (top of Figure 1) flow into a repository (center) where they are cataloged and graphically documented. The generated *structure*, *object* and *flow* diagrams are displayed to the user (center left). Under user control, the program units, one by one and their overall synthesis are translated into MODEL specifications (top right), which are stored in the repository. The user can then understand and modify the MODEL specifications using the Front-End tools (center left of Figure 1). The specifications are then translated into Ada (bottom right) which are also stored in the repository. They can then be functionally tested and integrated.

The MODEL specification consists of an abstraction of the logic in the program and thus helps in gaining understanding. The specification is translatable automatically (by the existing MODEL system) into C or Ada and also serves as the medium for testing, verification and maintenance. We propose

MODEL as the specification language because MODEL is a totally non-procedural language that expresses program function and algorithms through a set of diagrams, rules (in the form of equations) and declarations. The rules are readily readable. There are no side effects and no need to "think-like-a-computer" in order to understand them[2].

The new elements in Figure 1 are the translator from CMS-2 to MODEL, at the top right of the figure (discussed in Sections 3 and 4), and the front-end tools at the left center of the figure (discussed in Section 5). The other components are available off-the-shelf and are only briefly reviewed here. The CMS-2 to Ada translator is also described in detail in [15]. It was extended to use of shared memory, inter-task communications, and concurrent processing [16], which are essential in real time systems.

2. Computer Aided Software Engineering (CASE) Components

Table 1 lists the CASE components in Figure 1 for the CMS-2 to Ada Conversion.

The **Front-End** performs the communications between the user and the various tools.

The **Back-End** class consists of language processors that generate programs. These include the CMS-2 to MODEL and MODEL to Ada translators, as well as conventional compilers for CMS-2 and Ada. The CMS-2 to MODEL translation is the topic of section 3 of this paper. Note that the existing MODEL to Ada translator generates 100% of the Ada program, thus the user needs only to interact and debug MODEL specifications, not Ada or CMS-2 programs (further discussed below). Thus expertise in CMS-2 or Ada is not required from the user.

MODEL is a technology for automatically generating large-scale software. The MODEL system includes several primary components: a MODEL Configurator for "programming in the large", a system for timing analysis and design, a MODEL Compiler for "programming in the small", a MODEL Painter for specifying man-machine interfaces, and a MODEL Packager for arranging programs and data declarations into packages. Each component accepts as input a particular nonprocedural specification, analyzes it for logical correctness, schedules the order of events, and generates 100% of a highly optimized programs.

I FRONT-END		**III SUPPORT**
Expert Search	– Repository search and organization	Configuration Management
Diagramming	– Structure Diagrams	Document Generator
	– Dataflow Diagrams	Project Management
	– Object Oriented Diagrams	
	– Dependency Diagrams	**IV REPOSITORY**
Text Editors		Organization–Relational
II BACK-END		
CMS-2 to MODEL		***V WORKSTATION***
CMS-2 to :	Structure Diagrams	Common User Interface/Windows
	Dataflow Diagrams	Graphics
	Object Oriented Diagrams	High Computing Power
MODEL Configurator to Ada Multitasking Code		
MODEL Timing/Architecture Analysis and Design		
MODEL Package to Ada Package Code		
MODEL to Ada Compiler		
MODEL Painter		
CMS-2 Compiler		
Ada Compiler		

Table 1: CASE Subsystems.

The MODEL Configurator[7]: The user/designer shown at the center left of Figure 1 can draw dataflow diagrams via an interactive, graphical Front-End interface. Alternately, dataflow graphs, obtained from a CMS-2 system, can provide the starting point. They are called *configurations*. The dataflow diagrams are progressively "exploded" to show a hierarchy of configurations. The respective Ada programs are generated for each level of explosion. The Configurator identifies configuration inconsistencies, proposes missing declarations and definitions and checks for various ambiguities. When satisfied that the configuration is logically consistent, the Configurator optimizes opportunities for parallel execution of tasks and generates the multitasking control programs necessary to start and stop tasks, establish communications, and synchronize execution.

The initial partitioning into tasks, laid out along functional lines, may be inadequate in satisfying timing constraints and need re-designing. This approach encourages a spiral of partitioning and re-partitioning iterations. On each use of the Configurator, new multi-tasking programs are completely regenerated.

The Configurator provides automatic means of distributing a configuration across a network of heterogeneous processors. It schedules load-balances, monitors and migrates the execution of programs.

Timing Analysis and Design: Given the constraints of critical delays and the software architecture (the partitioning of the system into concurrent tasks) this system produces an optimized mapping of the software architecture to a hardware architecture. Given a hardware architecture and software architecture, it produces a realistic estimate of the critical delays.

The **MODEL Compiler** [6] incorporates knowledge of programming. It competes with expert human programmers in producing programs that are equal in their structure and efficiency. It produces programs in a variety of languages and for different computer architectures. The functions of the compiler are characterized as follows:

Tolerance of Omissions: To fully specify a problem so that its meaning is always expressed explicitly would be very tedious. Many details may be understood implicitly. A specification is typically an order of magnitude shorter than the generated program. This is necessary for a specification to serve also as an abbreviated abstracted documentation of the respective algorithms. The MODEL Compiler attempts to fill omissions whenever possible, alerting the user to the system's actions (warning messages) or soliciting from the user additional information (error messages). Common omissions that are automatically supplied are datatypes, data declarations, sizes of variables, and subscripting.

Checking: The checking is the major vehicle for debugging. Typically, the MODEL Compiler generates a program only after a number of interactions with the user, seeking common meaning and making changes. Our experience is that 70%-90% of the changes made in a specification throughout the development are stimulated by messages issued by the MODEL system; only 10%-30% are stimulated by the results of executing the generated programs.

Both filling in omissions in a specification and checking consistency are based on algorithms that construct and analyze a dataflow graph of the specification. The checking of consistency is based on the implicit redundancies in a specification. Thus, a specification is checked against itself rather than against separately stated assertions as is the case in verification.

Optimization: The MODEL system proceeds far more systematically and extensively, in global and local analysis and evaluation of alternatives, than a human programmer. Note that totally new, highly structured and optimized programs are generated for each set of maintenance changes in specifications.

Built-In Operations: Presently, there are over 100 such operations in MODEL, with the number increasing. There are operators available to perform matrix and relational algebra and solve sets of simultaneous equations, as well as to manipulate character strings. A user can define additional operations using the MODEL language

Code Generation: The MODEL Compiler incorporates knowledge of the facilities available in each target language. It employs chosen language constructs and semantics to accurately represent the meaning of a specification.

The MODEL Compiler can be used to generate programs that must be "plugged-in" to a system that may already exist or was not completely developed using MODEL. Generated specification programs may be the main module, functions or procedures.

The **MODEL Painter** [8] allows the user to specify reports or displays in the form of pictorial layouts which are translated into corresponding MODEL specifications. The specifications that represent the report/screen rules can be mixed as needed with additional processing rules. Again, all maintenance is performed on the pictorial layout.

The **MODEL Packager** allows for modularity in development by accepting in a package a collection of types, declarations, procedures, functions, or even other packages. The Package System generates complete Ada package code.

The **MODEL Library** is on a higher level than the Ada library. It contains the MODEL specifications organized in packages, same as in Ada. It is used for checking to assure that generated programs pass Ada compilation without any problems reported.

The **Repository** *is the key subsystem* that shares and retrieves all the information entered and needed in using the tools. Diverse information is stored in the repository.

3. CMS-2 Procedures To MODEL Translation

3.1. Overview

The translation from CMS-2 to MODEL is divided into two parts: in-the-large and in-the-small. Both parts are illustrated in Section 4. This section discusses the second part: translating procedures.

The translation is based on *equivalence* between the source CMS-2 program and the object MODEL specification. The equivalence maps the instances of assignments to variables in the source procedure into respective equations that define the variables in the object specification. The source procedure and the object specification are equivalent in the sense that the respective mapped variables have the same values.

The specification can be viewed as an abstract set of mathematical equations. It can also be viewed as a dataflow machine computational model. The computation finds values for all the variables which make all equations and declarations in the specification true.

We are not concerned whether the source procedure "makes sense", only that the source program is compilable and computable and the object specification variables have the same values as the respective variables in the source program.

The translation consists of a number of transformations. Starting with the source program, each transformation modifies a program into an equivalent program which is progressively closer to the object equational specification. Basically, the difference between a procedural program and an equational specification is as follows. Variables in a procedural program can have assigned to them none, one or several values in the course of sequential execution. In a mathematical equational specification, each variable can assume one and only one value. The transformations rename variables for each execution instance of program assignments so that each renamed variable has one and only one assigned value.

There are also simplification transformations that make the equational specification easier to understand and modify. They employ equivalence axioms and substitutions. Their objective is not only that the object specification is equivalent to the source program, but also that it is readable and understandable.

3.2. Transformations

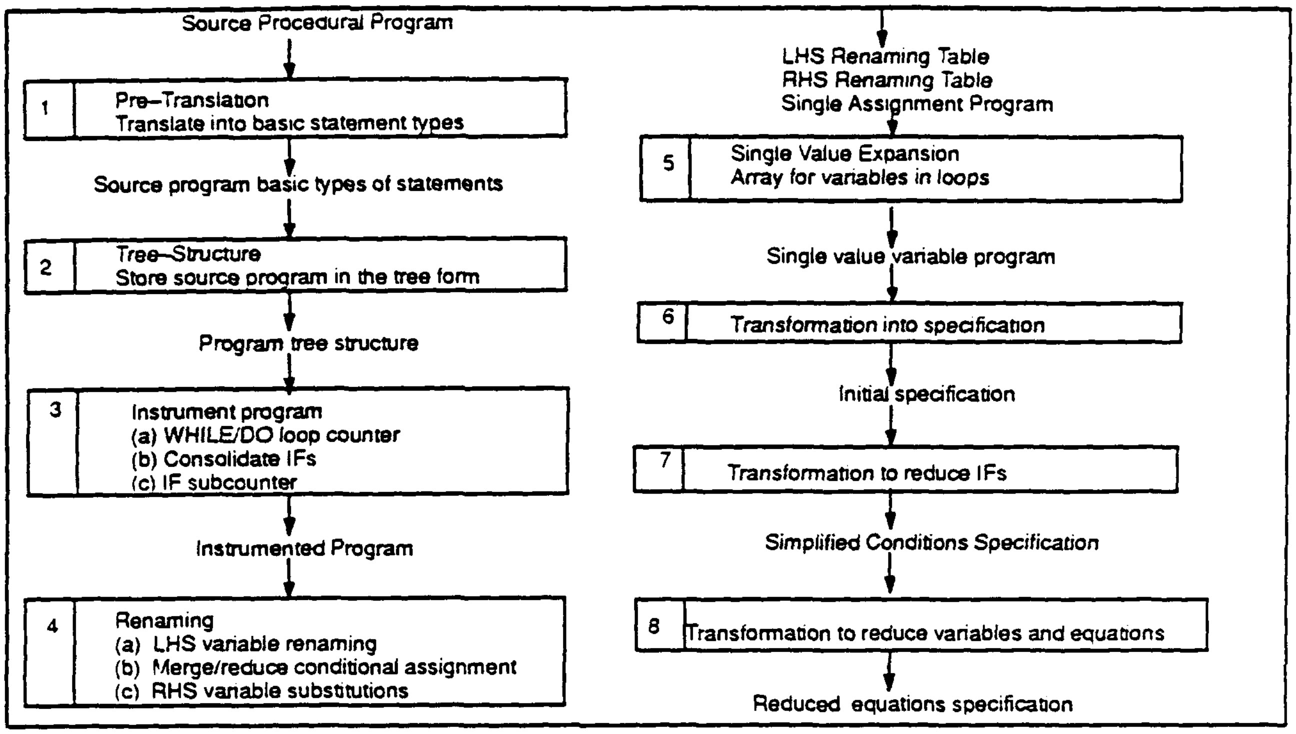
Figure 2 shows eight successive transformations (transformations 4 and 5 are multi-step). Following is a brief explanation of the need and purpose of each transformation. Each of the first five transformations provides as output a program that is equivalent to the input to the transformation. It starts with the source program and ends with assignments that can be directly transformed into equations The final program is translated into equations and declarations and then simplified in the second group of transformations. The transformations in the second group operate on equations. They use algebraic manipulations to collect like factors and make substitutions to simplify the specification.

Transformation 1 replaces subroutine calls, GOTO's, and references to shared storage with assignments, WHILE's and I/O statements[1]. Among all CMS-2 language components, we distinguish a basic subset which consists of variable declarations (with different data types and data structures), blocks (IFs and loops), assignments, and *Executive Service Requests* (ESRs). The declarations, blocks, and assignments have a common meaning to those used in other programming languages, although the syntax differs. The ESRs are operations performed by the Executive. They include I/O and interrupts. Non-basic components in CMS-2 are translated into the basic subset first and then translated into the specification. This produces a program utilizing only the basic types of statements.

Transformation 2 merely stores its source program in memory as a tree structure. The succeeding transformations scan the tree and modify it.

Transformations 3 and 4 are the most important. They are based on the following concepts:

- The program is viewed as a canonical tree of loop and IF nodes with assignment and declaration leaves.
- A repeatedly assigned variable in a loop is transformed into an array where each element can be referenced by providing its index.
- The last one of multiple assignments in a loop to the same variable is retained as a variable with a reduced dimension. This is done in order to simplify and localize the dependencies between variables in different loops.



Source Procedural Program
1 Pre–Translation
Translate into basic statement types
Source program basic types of statements
2 Tree–Structure
Store source program in the tree form
Program tree structure
3 Instrument program
(a) WHILE/DO loop counter
(b) Consolidate IFs
(c) IF subcounter
Instrumented Program
4 Renaming
(a) LHS variable renaming
(b) Merge/reduce conditional assignment
(c) RHS variable substitutions
LHS Renaming Table
RHS Renaming Table
Single Assignment Program
5 Single Value Expansion
Array for variables in loops
Single value variable program
6 Transformation into specification
Initial specification
7 Transformation to reduce IFs
Simplified Conditions Specification
8 Transformation to reduce variables and equations
Reduced equations specification

Figure 2: Program Transformations

Transformation 3 "instruments" the assignments in the program. Transformation 3 consists of three subtransformations. It introduces variables that count instances of assignments in a loop. Separate counts are computed for conditional assignments.

Transformation 4 also consists of multiple steps. It is illustrated schematically in Figure 3. It first gives distinct names to left hand side (LHS) variables of assignments. (e.g. in Figure 3, x is mapped into x1, x2 and x3.) To simplify later analysis, it merges conditional assignments to the same variable in a loop and reduces the dimensionality of arrays referenced outside the loops. Finally it substitutes for right hand side (RHS) variables in assignment statements expressions that use LHS variables. (e.g.it generates cond1, *cond2, and cond3 in Figure 3.) The result of this transformation is the* so called *single assignment* program.

Transformation 5 merely assigns separate storage space for each instance of variable assigned values repeatedly in a loop. They form an array of instances of the variables that are assigned values in a loop.

Transformation 6 essentially copies the assignments produced previously into a set of equations. It then generates declarations and header statements. The specification may now be longer than the source program, because a number of variables and conditions were added to make explicit all the interactions among variables (no "side effects"). Further analysis is conducted in later transformations to reduce the size of the specification.

Transformation 7 simplifies conditions in equations. Conditions are collected and factored and duplicates eliminated. Conditions are also simplified using equalities.

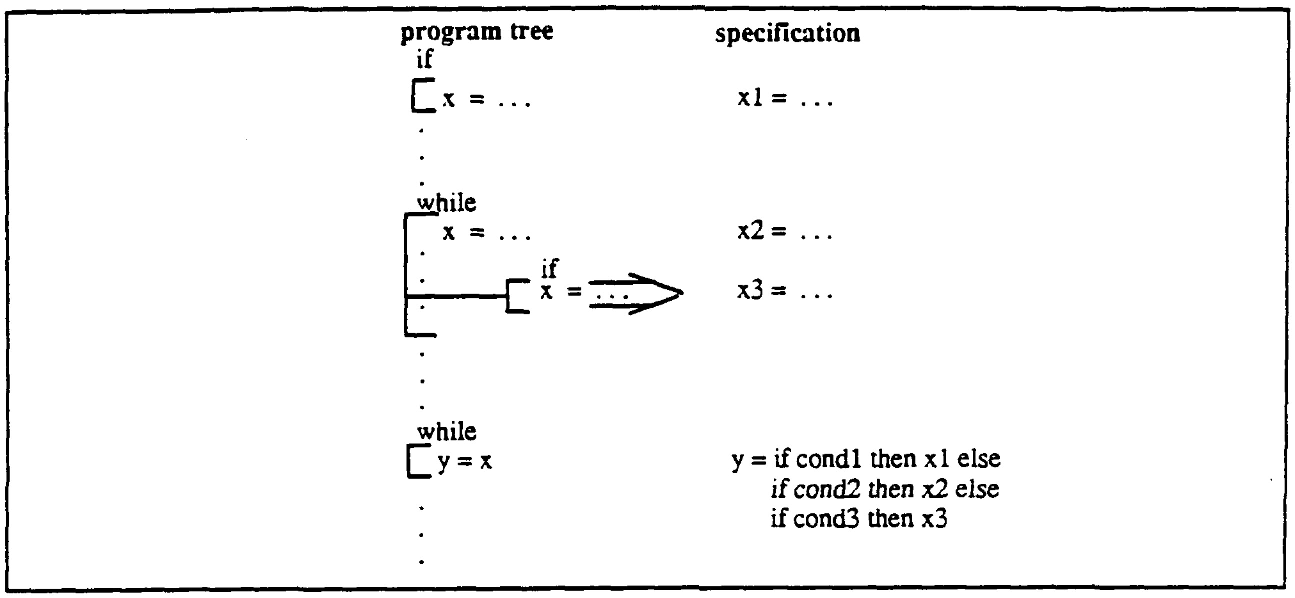


Figure 3: Illustration of LHS Renaming (x into x1, x2 and x3) and LHS conditions (cond1, cond2 and cond3).

Transformation 8 eliminates some variables by substituting their defining RHS expression in other equations.

The specification is shorter than the program in some aspects and longer in others. It omits all input/output, block and internal variable declaration statements of a program. It adds equations or declarations for certain subscripts and array sizes. It uses more elaborate conditions in some equations. However, the additions should help the understandability as they explicitly explain the program's "side effects" and point to exceptional cases that need to be considered for full understanding. Namely, a programmer (or a program generator) uses "side-effects" to optimize a program, but the result is more difficult to understand. The translation into a specification spells out the side-effects at a sacrifice of increasing the length.

This completes transforming the program into a *single value* form where each variable has one and only one value assigned to it. The resulting assignments can be viewed directly as equations.

The last three transformations in Figure 2 have the objective of producing a specification and simplifying it.

4. An Example

4.1. The Example CMS-2 Source Procedure

The example selected to illustrate the in-the-small translation and program reasoning is typical of procedures in real-time applications where tasks share memory. It is a CMS-2 procedures incorporating Peterson's algorithm to assure mutual exclusion of access by two tasks to a commonly shared memory. The Peterson algorithm provides a software solution to the mutual exclusion problem. The algorithm causes a task to wait until the other task unlocked the access. This example has been selected here because it is classical in illustrating use of shared memory. The problem is well known. It has evaded a solution until 1965 (Dekker's algorithm) and was improved in 1973 by Petersen. It is short and therefore can be easily used within the bounds of this paper. Most important, we will use proving its correctness (in Section 5) to illustrate improving understandability. The algorithm has been proven in a number of publications providing a means of comparison of ease of proof.

Figure 4 shows in diagrammatic form the in-the-large structure of a CMS-2 system called *overall*. It represents an example of the diagrams that are generated from translating the program. *Overall* contains the shared memory *shared* and a concurrent subsystem called *concrrent* that consists of two tasks *p1* and *p2*. Both tasks use the procedure *lock* to gain mutually exclusive access to a *critical section*, and the procedure *unlock* to allow access to the other task. Figure 4(a) shows the tree structure of the overall program. Figure 4(b) shows the object-oriented usage relationship among these components. Figure 4(c) illustrates the dataflow.

The CMS-2 program for *overall* is shown in Figure 5. The procedures, *lock* and *unlock*, used to illustrate in-the-small program reasoning are enclosed in frames in Figure 5 to emphasize them.

An explanation of the program in Figure 5 follows (also refer to Figure 4).

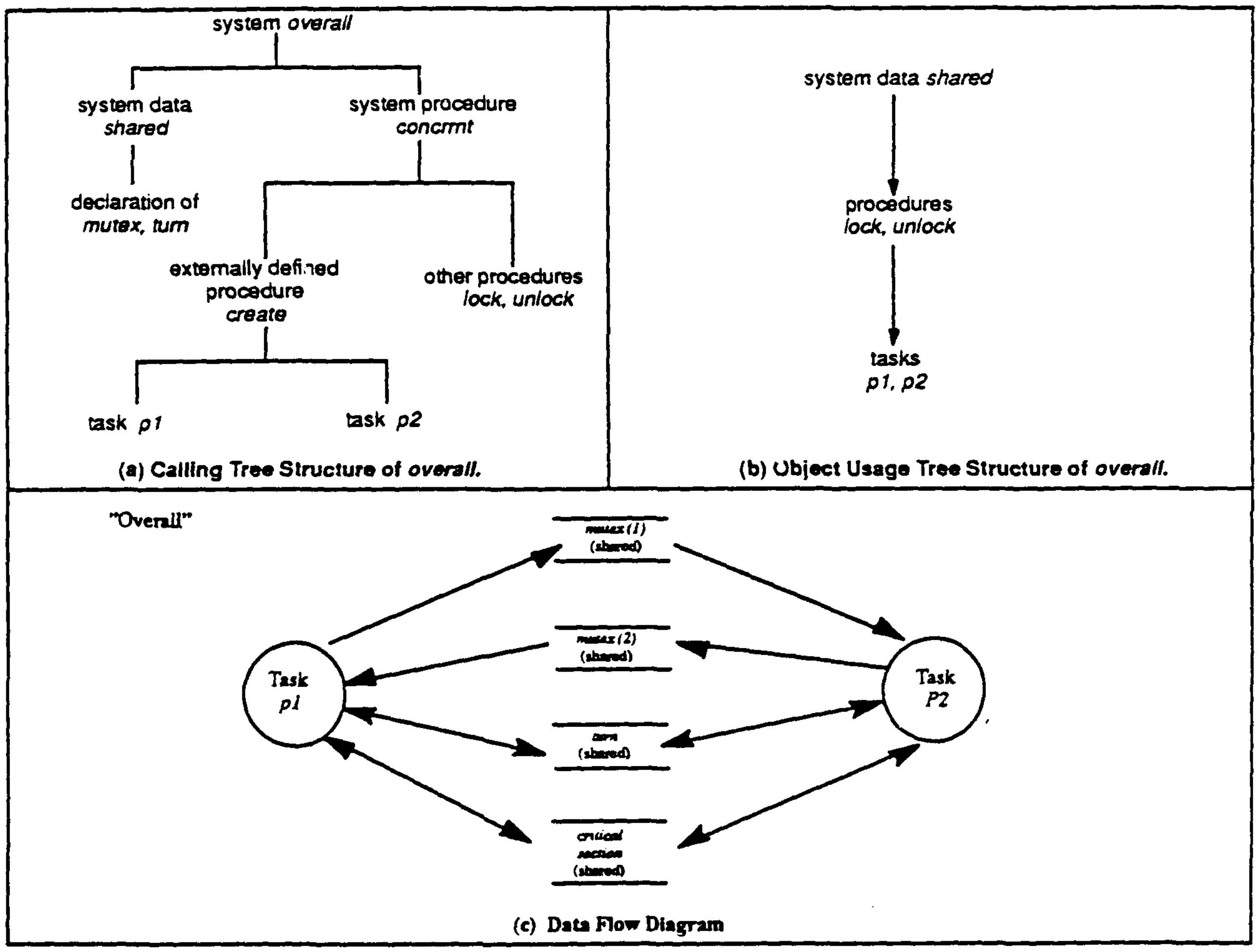

system *overall*
system data *shared*
system procedure *concrrnt*
declaration of *mutex, turn*
externally defined procedure *create*
other procedures *lock, unlock*
task *p1*
task *p2*
(a) Calling Tree Structure of *overall*.
system data *shared*
procedures *lock, unlock*
tasks *p1, p2*
(b) Object Usage Tree Structure of *overall*.
"Overall"
mutex(1) (shared)
mutex(2) (shared)
turn (shared)
critical section (shared)
Task p1
Task P2
(c) Data Flow Diagram

Figure 4: Diagrammatic Descriptions of *overall*.

```
overall SYSTEM $                "system name"
shared SYS-DD $                 "shared variable declaration"
    TABLE mutex V (B) 2 P 0 0 $
    END-TABLE mutex $
    VRBL turn I 8 U P 1 $
 END-SYS-DD $

concrrnt SYS-PROC $
 (EXTDEF) PROCEDURE create $   "executive creating "
  esan INPUT p1 $              "concurrent tasks"
  esan INPUT p2 $
 END-PROC concrrnt $

+========================================+
| PROCEDURE lock INPUT i $               |
|  SET mutex(i) TO 1 $                   |
|  SET turn TO i $                       |
|  VARY UNTIL (COMP mutex(3-i) OR turn EQ 3-i) $ |
|  END $                                 |
| END-PROC lock $                        |
+========================================+
| PROCEDURE unlock INPUT i $             |
|  SET mutex(i) TO 0 $                   |
| END-PROC unlock $                      |
+========================================+
```

```
PROCEDURE p1 $              "task 1  "
  " non-critical section "
  lock INPUT 1 $
  " critical section "
  unlock INPUT 1 $
END-PROC p1 $

PROCEDURE p2 $              "task 2  "
  " non critical section "
  lock INPUT 2 $
  " critical section "
  unlock INPUT 2 $
END-PROC p2 $

END-SYS-PROC concrrnt $

END-SYSTEM overall $
```

Figure 5: ***overall*** **Concurrent System with Mutual Exclusion Procedures** ***lock*** **and** ***unlock.***

The *shared* SYS-DD statement in Figure 5 declares a structure which is shared by all the tasks in the system. It consists of a two element array called *mutex* which is used to express intention of each of the respective tasks to enter the critical section (initialized to zero) and a variable called *turn,* which indicates which of two tasks (p1 or p2) has the lower priority to enter the critical section (initialized to 1). As shown in *procedures p1* and *p2,* prior to entering a critical section, they call a procedure *lock* with its own index *i* (for *p1, i*=1; for *p2, i*=2). Procedure *lock* consists of first expressing the intention to enter the critical section (SET mutex(i) TO 1) at lower priority for entry (SET turn TO i). Next there is a loop in which the procedure waits until either the other task has no intention to enter the critical section (*mutex*(3-*i*)=FALSE) or if the other task has lower priority (*turn*=3-*i*). The unlock procedure records in the shared memory variable *mutex*(i) that the task has left the critical section.

4.2. The Corresponding MODEL Specification

Figure 6 shows the MODEL specification generated from the *lock* and *unlock* procedures of Figure 5. The specification for *lock* in Figure 6(a) is explained below.

Header: The header and the declaration of input and output data constitute the interface between the outside world and the procedure. The header consists of three statements. The first gives the procedure *name* *(lock)* and its parameter (*i*). Also it states that the procedure uses an EXTERNAL memory *shared* which in this case is shared by the *lock* procedures in the two tasks. The structure of *shared* in the program is also shown in Figure 7(a).

The SOURCE statement in Figure 6(a) specifies the input parameters *i* and the structure *sharedin.* The TARGET statement gives the output structure *sharedout*.

Next are the declarations of the structures of input/output variables.

```
/* INTERFACE */
   /* HEADER */
   PROCEDURE: lock (i) EXTERNAL(sharedin, sharedout);
   SOURCE: i,sharedin;
   TARGET: sharedout;

   /* INPUT/OUTPUT DECLARATIONS */

   i IS FIELD (NUM 1);

   1 sharedin IS FILE RENAME AS shared ORG: SHARED,
     2 mutexr(*) IS RECORD RENAME AS mutexr,
     3 mutex_7(3-i)IS FIELD(BIT 1) RENAME AS mutex(3-i),
       3 turn_8 IS FIELD(NUM 1) RENAME AS turn;

   1 sharedout IS FILE RENAME AS shared ORG: SHARED,
     3 mutex_2(i) IS FIELD(BIT 1) RENAME AS mutex(i),
     3 turn_4 IS FIELD(NUM 1) RENAME AS turn;

/* EQUATIONS */

   mutex_7(sub1,3-i)=DEPENDS_ON(turn_4);
   END.mutex_7(sub1,3-i) = ^mutex_7(sub1,3-i) |
                                    turn_8(sub1)=3-i;
   turn_4 = i;
   mutex_2(i) = TRUE;
```

(a). The Specification For lock

```
PROCEDURE: unlock(i) EXTERNAL(shared);

  TARGET: shared;
  SOURCE: i;

i IS FIELD(BIN FIX);

1 shared IS FILE ORG: SHARED,
  2 mutex(2) IS FIELD(BIT 1);

  mutex(i) = FALSE;
```

(b). The Specification For unlock

Figure 6: The Specifications Generated From The CMS–2 Program For *lock* and *unlock* in Figure 2.

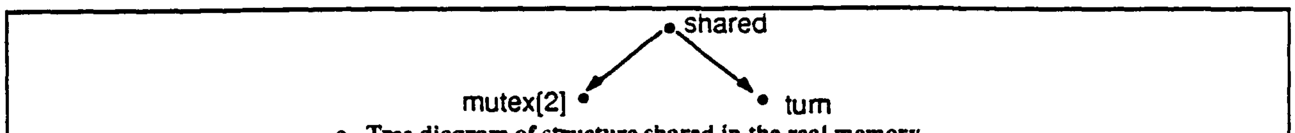
i is an input parameter. It is an integer for representing the two tasks *i*=1 or *i*=2.

Shared memory in a MODEL specification is envisaged as follows. From the point of view of the procedure *lock*, *shared* is envisaged similar to an external input or output device. It reads or writes values from and to the shared memory sequentially. Therefore the shared memory appears to the *lock* procedure as if it was read from and written to as a sequential device. *sharedin* and *sharedout* are then two EXTERNAL I/O devices through which *lock* "sees" the *shared* memory. This view of shared memory has been developed to extend the procedural-to-equations translation to concurrent programs.

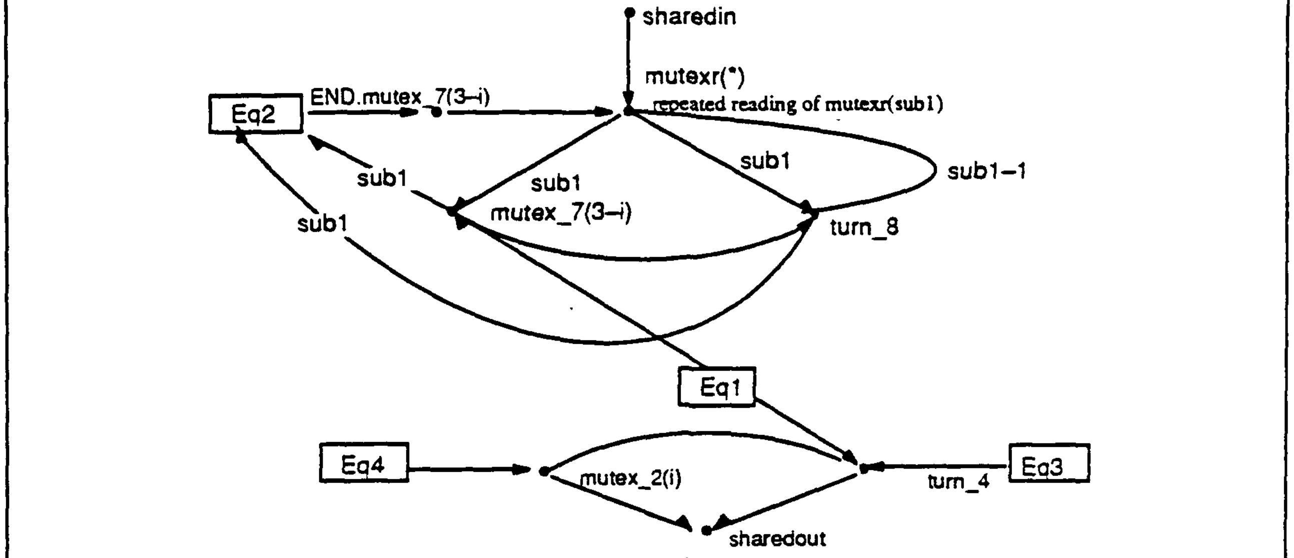
Every shared variable which is read or written repeatedly is given a different name. Thus sharedin consists of a vector of records *mutexr*. The unknown number of repetitions is indicated by *. Every instance of *mutexr* contains mutex_7(3-i) followed by turn_8. (The suffixes indicate the line number in the program of the assignment where the variable is used on the LHS.) *sharedin* is of the SHARED organization, i.e. memory shared among tasks. It is mapped onto the *shared* structure by the RENAME clauses, i.e. *sharedin* is mapped onto *shared*, *mutex_7(3-i)* is mapped onto *mutex(3-i)*, and *turn_8* is mapped onto *turn*. In the generated program, the program variables are read in place of specification variables.

sharedout declarations follow a similar approach and meaning to that explained above for *sharedin*.

Next are shown in Figure 6(a) the four equations. They are referred to in the following as Eq1 to Eq4. Inherent in each equation is a precedence relationship, namely precedence of the RHS variables over the LHS variables. These dependencies are illustrated in a dataflow diagram form in Figure 7(b). Eq1 merely indicates precedence of writing turn_4 over all repeated readings of *mutex_7(sub1,3-i)*; sub1 is a subscript, i.e. a variable index of *mutex_7*; *3-i* is a scalar.



a. Tree diagram of structure shared in the real memory.



b. Dataflow diagram showing the array graph view.

Figure 7: Graphic Representation of the MODEL Specification in Figure 6.

Eq2 defines a new variable *END.mutex_7(sub1,3-i)*. The prefix END indicates that this is a boolean vector of the same shape as the suffix variable. This variable has a value FALSE for all elements of the vector, except the last element which has a value TRUE. Namely, this variable provides a way for denoting the size of the vector.

Eq3 and Eq4 define the values of the two variables mutex_2(i) and turn_4 in the output (*sharedout*).

A MODEL specification has a graphic representation called *array graph* that helps the visualization of the algorithm. It is used to explain the specification and reason about it. It is illustrated in Figure 7b for the example of Figure 6(a). It is called an *array graph* because the nodes and edges can represent entire arrays. It has nodes for variables (denoted by dots) and for equations (denoted by rectangles). Otherwise, it may be interpreted similar to Petri-Nets. Namely, a node at one end of an edge must be defined (input (read), output (written) or the equation evaluated) preceding the node at the other end. The nodes and edges are marked with subscript expressions to denote the appropriate precedence correspondence of respective elements of the arrays at the ends of the edge. The precedences in the array graph may be seen from the way that the MODEL to Ada translator orders the nodes in generating a procedural program. Figure 8 illustrates the order of program events in generation of a program by the MODEL to Ada translator from the array graph in Figure 7(b). It also, illustrates the order of execution of nodes in Figure 7(b).

Actually, further optimization in generating a program for the specification finds that it is sufficient to use the program scalars for the specification vectors.

```
sharedin
Eq4
Eq3
mutex_2(i) write
turn_4 write
sharedout
loop for sub1 while^END.mutex_7 (sub1, 3-i)
        Eq1
        mutexr(sub1) read
        mutex_7(sub1,3-i)
        turn_8(sub1)
        Eq2
        END.mutex_7(3-i)
end loop
```

Figure 8: Order of Events In a Program Generated for the Array Graph of Figure 7.

5. Verification Based On The Specification

The understandability of the specification, as compared with that of the program, is an important issue in a reverse software engineering of an application. The ease of verification of the specification vs. the program is an indicator of ease of understanding. Program verification methodology consists of two sets of statements. On the one hand there is the program that fully expresses the inherent algorithm as well as its functional behavior. On the other hand there is a set of assertions that have been composed completely independently of the program. The same approach is used here except that the specification is used in place of the program. The assertions may be derived based on a requirement document or reflect functional behavior. Human insight is necessary to compose the assertions. The essence of verification is to prove behavior consistent with the assertions[4,11,12].

The approach here replaces the program with its generated specification[17]. There are also other important differences.

1. Program verification involves also use of a program schema. We use instead the array graph.
2. In our approach, both the assertions and the specification use declarative, non-procedural mathematical semantics. In program verification, the program and assertions use different semantics.
3. Our approach is less complex because it is sufficient to analyze separately each of the concurrent programs and its shared memory, while verification of concurrent programs using Temporal Logic[9] analyzes the interleaved concurrent programs. For example, referring to Figure 4(c), it is sufficient to analyze *p1*, *sharedin* and *sharedout*, and we need not analyze the interleaved *p1* and *p2*.
4. Algebraic manipulation (using all axioms based on equivalence) can be used to prove consistency between the assertions and the respective specification. Verification based on program requires tracing the program schema.
5. Another benefit is that it is possible to generate from the MODEL specification a well structured and optimized program, which may be superior to the original program.

For these reasons, *specification verification* is claimed to be easier than *program verification*.

Generally, the approach to specification verification is as follows. First it is necessary to have insight into how variables and equations in the specification express entities and relations in the requirements. This requires human insight. This approach is illustrated in the following.

There are three requirements of the procedures *lock* and *unlock* that need to be reasoned and verified. They are referred to as *mutual exclusion*, *bounded-wait* and *progress*. They are explained and verified below.

Mutual exclusion: This requires that only one task at a time can use the critical section. This requirement of *lock* can be expressed as follows.

i. If one of two tasks is in the critical section then the other task must wait.

ii. If both tasks attempt entry to critical section simultaneously, only one task is allowed to proceed while the other must wait.

In terms of the specification for *lock*, the end of waiting is expressed by the variable *END.mutex_7(subl,3-i)*. Every integer value of subl, i.e. 1, 2 . . ., implies a waiting period. The value FALSE for this variable means continued waiting, while attaining the value TRUE means the last element of mutex_7(subl, 3-i) followed by access to the critical section. This variable is defined by the equation (Eq2)

```
END.mutex_7(subl,3-i)=^mutex_7(subl,3-i)
                     | turn_8(subl)=3-i;
```

The first requirement is satisfied in *lock* when *mutex_7(subl,3-i) = TRUE* (when the other task is in the critical section) and the procedure *lock* has previously set *turn_4 to i*. Therefore END.mutex_7(subl,3-i) = FALSE.

The second requirement is satisfied as follows. When *lock* is called by both tasks simultaneously, *mutex_7 (subl, 3-i)* is then TRUE for both tasks, (i.e. for i=1 and i=2) but *turn_8(subl)* can be TRUE only for one of the tasks, never for both, as *turn_8* is RENAMED *turn* and it can have a value of 1 or 2 but not both values simultaneously. Thus *END.mutex_7(subl,3-i)* can be TRUE for only one task which enters the critical section while the other task waits.

Once the values of mutex_7(subl,3-i) and turn_8(subl) have been asserted to correspond to the requirement, END.mutex_7(subl,3-i) can be evaluated symbolically automatically. Alternatively, the user can find it using substitutions, aided by display of the array graph. This is further illustrated by the vectors of these variables in Figure 9.

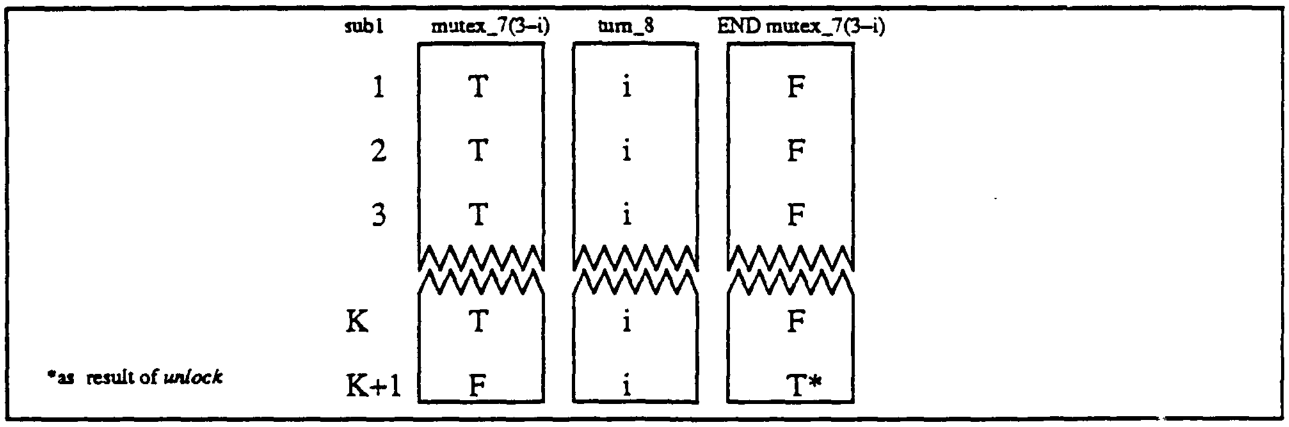


Figure 9: Illustration of Values of Variables for Verifying Mutual Exclusion.

Bounded wait: This requirement states that once a task has left the critical section, if the other task has been waiting, it is next given access to the critical section. The waiting task evaluates *END.mutex_7(subl,3-i)* as FALSE repeatedly for each incremented subl. The other task calls *unlock* that sets mutex(i) = FALSE (see specification for unlock, Figure 6(b)). Then for the first task, *mutex_7(subl,3-i) = FALSE* and END.mutex_7(subl,i) = TRUE. It enters the critical section. The other task can not re-enter the critical section (no matter how fast it is) as upon entry it sets *mutex_2(i)* to TRUE and *turn_4* to *i*. Therefore its *turn_8(subl)=3-i* is FALSE and its END.mutex_7(subl,3-i) is FALSE.

Progress: This requirement means that one task can always enter the critical section if the other task has not expressed the intent to do so. The proof of this requirement is a special case of the proof for Bounded-wait. It corresponds to setting *mutex_2(i)* to *TRUE* by one task (through call of procedure *unlock*). This allows the other task to enter the critical section by calling procedure *lock*.

6. Conclusion

In conclusion we want to reiterate the following points.

Reverse Software Engineering is a method to utilize outdated programs to reduce cost of developing new replacement systems. The emphasis is on *reducing cost* of *replacement systems*. The old systems are assumed to be inadequate in their structure, functionality or implementation technology. Still, to reduce cost it is desired to find and reuse what is available in the old system as a basis for making appropriate changes, deletions and additions. This is becoming enormously important because of the immense investment in software and the fast aging of systems due to the rapid introduction of new technology.

Mathematical representations of programs have been widely claimed to be advantageous for understanding, checking and modifying software. Translation into a mathematical representation has been the constant theme in research into a number of directions concerning procedural programs. The underlying notion here is to use a mathematical representation as an intermediate step in Reverse Software Engineering as well as in new software development. It is proposed as the medium for understanding, analyzing and changing old programs as well as the medium for new software development. Many of the mathematical representations of procedural programs proposed in the past involved unfamiliar syntax and semantics which would make it difficult for uninitiated users to employ them for maintenance. The choice here has been to use widely known regular and boolean algebras for mathematical representation. We have extensive experience with the MODEL system in using equational specifications for software specification, testing, maintenance and code regeneration.

The example used shows extending the algorithm to concurrent programs.

Because of scope limitation, we have not discussed the question of optimization in regenerating the program from the specification. This is a large and complex topic in itself. The optimization is typically performed in multiple levels, starting with the level of global program design and progressively focusing on more local optimization. The global optimization is well advanced and we and others are continuing research on local optimization of procedural programs. There is also much research on optimization in parallel programs. The results to date and future prospects are good for highly satisfactory optimized program regeneration.

7. References

1. E. Ashcroft, Z. Manna, ''The Translation of Goto Programs to While Programs,'' *Proceedings, IFIP* Congress 1971, North-Holland Publ. Co. Amsterdam, pp. 250-255, 1972.
2. J. Backus, ''Can Programming be Liberated From the Von Neumann Style? A Functional Style and its Algebra of Programs,'' *Communications of the ACM*, V21 #8, August, 1978.
3. J. Baron, B. Szymanski, E. Lock and N. Prywes, ''An Argument for Nonprocedural Languages,'' *Proc. Workshop Role of Languages in Problem Solving-1*, 1985.
4. Arthur Bernstein, Paul K. Harter, Jr. ''Proving Real-Time Properties of Programs with Temporal Logic,'' *ACM*, 1981.
5. T. Cheng, E. Lock and N. Prywes, ''Use of Very High Level Languages and Program Generation by Management Professionals,'' *IEEE Transactions on Software Engineering*, V10 #5, September 1984.
6. Computer Command and Control Company, ''The MODEL Compiler Usage and Reference Guide - Non-Procedural Programming for Non-Programmers,'' 2300 Chestnut Street, Philadelphia, PA 19103, 1989.
7. Computer Command and Control Company, ''The MODEL Configurator Usage and Reference Guide - Non-Procedural Programming for Non-Programmers,'' 2300 Chestnut Street, Philadelphia, PA 19103, 1989.
8. Computer Command and Control Company, ''The MODEL Painter Usage and Reference Guide - Non-Procedural Programming for Non-Programmers,'' 2300 Chestnut Street, Philadelphia, PA 19103, 1989.
9. Fred Kroger, ''Temporal Logic of Programs,'' EATCS Monographs on Theoretical Computer Science, Vol. 8, 1987.
10. K.S. Lu, "Program Optimization Based on a Non-Procedural Specification", Ph.D dissertation, Department of Computer Science, University of Pennsylvania 1981.
11. Z. Manna, ''Mathematical Theory *of Computation*,'' McGraw Hill Book Company, 1974.
12. Amir Pnueli, ''The Temporal Semantics of Concurrent Programs,'' *Semantics of Concurrent Computation*, Proceedings of International Symposium, Springer, 1979.
13. N. Prywes and A. Pnueli, ''Compilation of Nonprocedural Specifications into Computer Programs,'' *IEEE Transactions on Software Engineering*, V9 #3, May 1983.
14. N. Prywe˙ X. Ge, I.Lee and M. Song, ''Reverse Software Engineering,'' Research sponsored by the Air Force Office of Scientific Research, Contract AFOSR-88-0116, University of Pennsylvania, December 1989.
15. M. Song, ''Reverse Software Engineering of Concurrent Real Time Programs,'' Master thesis for the Department of Computer and Information Science, University of Pennsylvania, Contract AFOSR-88-0116, December 1989.
16. B. Szymanski, E. Lock, A. Pnueli and N. Prywes, ''On the Scope of Static Checking in Definitional Languages,'' *Proc. of the ACM Annual Conference*, San Francisco, CA, pp. 197-207, October 1984.
17. B. Szymanski, N. Prywes, ''Efficient Handling of Data Structures in Definitional Language,'' Science of Computer Programming, pp.221-245 No. 10, October 1988.

A SCHEMA-BASED APPROACH TO SOFTWARE SPECIFICATION DERIVATION

Mehdi T. Harandi and Kanth Miriyala
Department of Computer Science,
University of Illinois,
1304 W. Springfield Avenue, Urbana, IL 61801
Telephone: (217) 244-5915

Abstract

Techniques and tools are needed to assist humans in the arduous task of writing formal software specifications. In this paper, we describe an approach based on the use of schemas to derive formal specifications from (semi-) informal descriptions. Issues such as abstraction-level of schemas, internal representation of informal descriptions, and organization of the knowledge base of schemas, are discussed. The paper also addresses the process of schema retrieval, instantiation, and refinement. The derivation process is illustrated using an example.

1 INTRODUCTION

The process of deriving programs from their specifications is well understood. There is a large corpus of knowledge in the literature about transformations necessary to synthesize programs. However, at present, our understanding of the nature of transformations underlying the process of specification derivation is not adequate to meet the needs of automation. Recently transformational rules have been formulated that are useful in restructuring, completing, or refining a given (preliminary) formal specification [3,4,5,6,12,13,14]. But we are interested in a computational model for deriving formal specifications starting with a more informal problem description (e.g., in a restricted subset of the natural language).

Our computational model is based on the model used by human experts. Human experts rarely write formal specifications from scratch. Instead they identify familiar chunks in the informal description and reuse, modify, or combine known specification chunks to derive a specification for the given problem. Similarly, our prototype, SPECIFIER, uses a knowledge base of chunks from past experience to derive formal specifications. Chunks of past experience are stored:

a. in their concrete or raw form; or

b. at some higher level of abstraction (*schemas*).

Given a new problem, relevant schemas or analogous past problems or both are used in deriving a formal specification. We call the use of knowledge in form (a) as the *analogy method* and the use of knowledge in form (b) as the *schema-based method.* A description of the analogy method can be found in [15,16]. In this paper we discuss the issues involved in the schema-based method and describe the prototype, SPECIFIER. The system uses a restricted subset of the natural language (based on an EBNF grammar) as the informal specification language and a slight modification of PLUSS [2] as the formal specification language.

Efforts in providing automated assistance with the process of specification derivation have mainly been concerned with evaluation, critique or enhancement of informal descriptions. One of the first such efforts was that of Balzar et al [1]. The prototype that they developed is a domain-specific tool that uses extensive context information to correct and complete a given partial description. The Requirements Apprentice [17,18] uses clichés and general purpose reasoning to derive a validated requirements document from an ambiguous, incomplete, and inconsistent specification. This system uses extensive domain knowledge. For example, to derive a library specification, it uses knowledge about information systems, tracking systems, and repositories. Fickas's system, KATE [8,9,10], critiques and modifies formal specifications presented to it using domain-specific knowledge. Feather [6,7] and Johnson [12,13,14] describe transformational approaches toward incremental construction of formal specifications. Feather describes results of hand-performed experiments. Johnson's system provides several high-level editing commands that help in applying transformations. However, the user is responsible for choosing appropriate editing steps.

Our work is similar in spirit to the specification/requirements assistant proposed by Green et al [11] as part of their Knowledge-Based Software Assistant (KBSA). However, our work focuses on the *specification phase* which succeeds the *requirements analysis phase.* In the requirements analysis phase, the requirements analyst obtains the needs of the users during a "skull session". After this the analyst prepares an informal requirements document. From such a document, specification is derived by a process of formalization of requirements. KBSA is aimed to be a domain-specific system. In contrast, SPECIFIER is a domain independent specification derivation system which accepts informal descriptions (like those in an informal requirements document) in a restricted subset of the natural language. It makes all the decisions that arise during the specification process and prompts the user for additional or missing information. It provides a uniform and integrated approach to deriving structured formal specifications using schema-based and analogy methods. The schema-based approach utilizes domain independent knowledge of commonly occurring operations. Thus like KBSA, our system is capable of providing assistance in various phases of specification derivation. However, our work focuses on the formalization of specifications and sidesteps other issues like consistency and completeness checking, requirements tutoring, and prototype testing which are proposed for KBSA.

2 ISSUES

Given a library of specification schemas and proper methods for indexing and accessing this library, the process of deriving formal software specifications by the schema-based approach can be summarized as follows:

- Convert a given informal description into an internal representation.
- Using the internal representation, find a schema that is most appropriate.
- Instantiate the retrieved schema to obtain a specification fragment for the given problem.
- Define (recursively) any non-primitive predicates or non-primitive data types used in the formal specification fragment derived above.
- Put together the derived specification fragments

Three basic issues are involved in the above process:

1. schema representation;
2. internal representation of informal specification; and
3. organization of schemas.

2.1 SCHEMA REPRESENTATION

There are several levels of abstraction at which schemas can be defined depending on how many operations are collected together in a schema. One possible way is to define a library of parameterized specifications for data types. Each parameterized specification contains a collection of operations. For instance, "LIST[ELEMENT]" could be a parameterized specification with ELEMENT as a parameter and *Create*, *Insert*, *Append*, and *Delete* as operations. The formal parameter ELEMENT itself could have certain conditions imposed on it. These conditions should be satisfied by the data type which is the actual parameter. A user obtains a formal specification simply by choosing a parameterized specification and instantiating it with appropriate actual parameters. While such a library is useful and has a simple access and use mechanism, the form of specifications is too rigid to be generally applicable. For instance, a parameterized specification of the list data type may entail that lists are constructed incrementally starting with the empty list and successively adding elements to it. This rigid definition prevents its use in situations where a non-empty list is required (e.g., to generate *names* which are non-empty lists of characters).

The main drawback of the above representation is that abstractions of entire data type specifications are stored as schemas. It is rare that one would want exactly the same set of operations. To improve reuse we descend to the *operation-level*; that is, we represent schemas as abstractions of operations rather than datatypes. Then the derivation of a specification involves instantiation of several such operation schemas to obtain operation definitions and putting together the resulting definitions to obtain a specification of the data type or program in question. For example, instead of instantiating the CREATE schema to return an empty list, it is instantiated to return a non-empty atomic element (to generate *names*).

However, even at the operation-level there exist a gamut of possibilities for representing schemas. At one end of this spectrum would be schemas representing a generic conditional structure of the specification and at the other end are schemas representing a specific operation of a specific datatype. The former is too general; that is, it would be extremely hard to find out what the conditions should be and to reason about actions to be performed depending upon the condition. The latter would work well if we were to derive exactly the same operation but would be of little use for anything else. Hence, we decided to choose an operation-level of schema representation that lies somewhere between these two extremes. In such a representation, each schema is associated with a concept which represents a class of related operations. For example, the *remove* schema associated with the remove concept, can derive definitions of operations involving removal of an element from a collection or structure. A few other examples of schemas are *create*, *add*, *retrieve*, *replace*, *size*, and so on. Often we define more than one schema for each such class. For instance, it is convenient to define two schemas for *add* class of operations, depending upon what kind of add operation we want (i.e., constructor or modifier).[1]

In our prototype, each schema has the following information encoded in it. An example of a schema is given in section 4.

[1]The type that is being defined is called the *type of interest* (TOI). A smallest set of operations that can derive all possible instances of the TOI is the set of *constructors* or *generators*. The operations that return TOI but do not belong to constructors are called *modifiers*.

- **Preconditions:** A set of preconditions is associated with each schema. Each precondition is either a boolean condition or an information requirement. If a condition is not satisfied, the schema cannot be applied. If an information requirement is not met, processes are initiated to obtain such information. For example, the preconditions of the *remove* schema state that the constructors should be available prior to its application. If the constructor have not already been defined, a process is initiated to derive specifications for the constructors first.

- **Axiom schemes:** Axiom schemes are a set of abstract representations of axioms. The operation names and arguments to operations are schematic. Appropriate instantiation of these schema variables results in axioms for the current problem.

- **Instantiation rules:** These rules state how the various predicates, operation names, and other variables used in the axiom schemes are instantiated.

2.2 INTERNAL REPRESENTATION OF INFORMAL SPECIFICATION

The user describes the problem to be formally specified in a restricted subset of the natural language. The informal specification has to be presented to the system operation by operation. For example, the operation to replace the first occurrence of an integer x with integer y in an array $A[1:n]$ is described as:

operation: replace
input: integer (x); integer (y); array [1:n] of integer (A);
output: array [1:n] of integer(A')
constraint: The first occurrence of x in A is replaced with y

The operation definition is converted into an internal representation, which should satisfy the following requirements:

- The representation should organize the information in the informal specification in such a manner as to enable derivation of structured formal specifications.

- It should make the process of schema-based derivation of formal specifications easy.

- It should also enable incremental acquisition of the informal specification.

We have chosen a tree-based representation method called *structure trees* to represent informal specifications internally. In this form of representation, each node represents a concept and its children nodes represent inputs, outputs, and conditions on the concept.

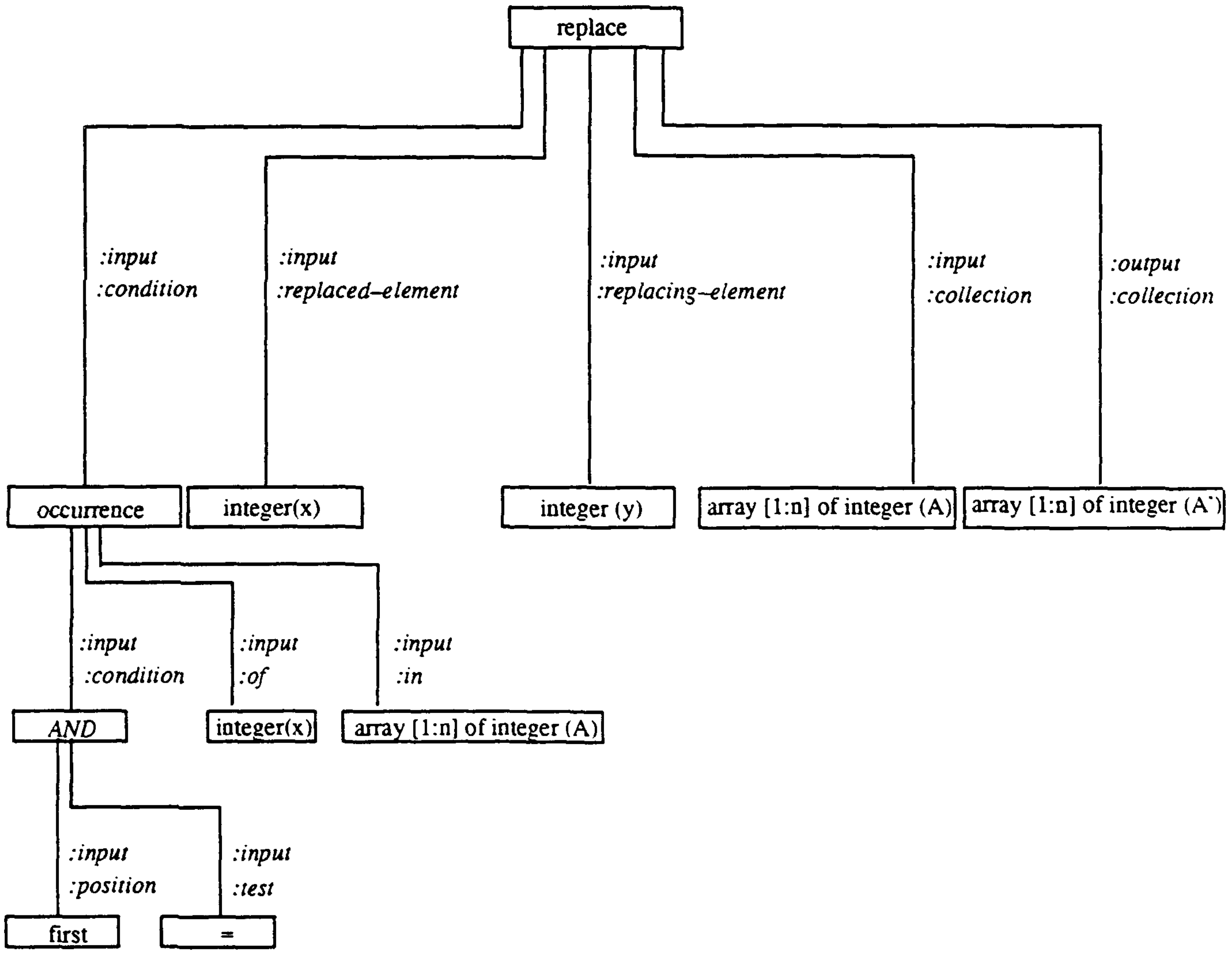


Figure 1: Structure tree for replace operation

Each of the inputs, outputs, and conditions could in turn be a structure tree. The structure tree for the informal specification of *replace* is shown in Figure 1.

The concepts closer to the root in a structure tree are more abstract and are incrementally refined in the lower levels. Thus the structure tree represents the information in the informal specification in an abstraction hierarchy. The formal specification for the structure tree is derived in a top-down manner and the derivation can consist of several passes. Each pass defines a part of the structure tree (with the first pass defining a part containing the root of the structure tree). Non-primitive datatypes and predicates present in the current formal specification are specified next. This process continues until all the datatypes and predicates have been defined in terms of primitives (i.e., basic data types and predicates). Thus, structured specifications are obtained. Moreover, for any structure tree, its root concept can be used to retrieve associated schemas from the concept dictionary, and its children can be used to instantiate retrieved schemas. Lastly, structure trees can be incrementally extended by adding structure trees for non-primitive predicates. For each non-primitive datatype, the user is prompted, informal specifications are obtained, and formal specifications are derived.

2.3 ORGANIZATION OF SCHEMAS

Once the representations for schemas and structure trees have been chosen, the memory should be organized to allow efficient access of appropriate schemas using structure trees. A simple scheme works very well.

The long-term memory or *concept dictionary* is a huge set of concepts. This set is partitioned into classes, with each class containing a set of synonymous concepts. Members of each class point to a special member called the *representative concept*. Schemas are associated with the representative concepts. For example, *remove*, *delete*, and *pop* are considered synonymous with (say) *remove* as the representative concept, having a pointer to *remove* schema.

3 THE DERIVATION PROCESS

The formal specification of an operation is derived from the structure tree of the operation in a top-down manner. For each undefined, non-primitive concept, beginning with the root of the structure tree, the following schema selection and instantiation strategy is used.

1. Retrieve the representative concept for the undefined, non-primitive concept.

2. Apply the schema selection knowledge associated with the representative concept to select an appropriate schema.

3. Check the preconditions of the schema. If some preconditions are not satisfied, spawn processes to try to satisfy them. For instance, if the precondition states an information requirement, the spawned process may simply prompt the user to provide the missing information. On the other hand, if the precondition states that constructors need to be defined, the spawned process initiates their specification derivation.
4. When the preconditions are satisfied, instantiate axiom schemes using the corresponding instantiation rules.
5. Simplify the axiom definitions formed using metarules. Metarule application may need information related to previously defined operations, e.g., postconditions of constructors may be needed to evaluate the truth of a condition. It might also be necessary to obtain information regarding the nature of relationship between certain concepts from the concept dictionary.

4 AN EXAMPLE

Suppose that we want to derive the formal specification of the `replace-first-occurrence` problem stated in section 2.2.

The preprocessor parses this informal specification. Important concepts are extracted from the *constraint* part. Structure templates are retrieved for them (from the concept dictionary), filled, and assembled to obtain the structure tree shown in Figure 1.

Schema-based derivation method is invoked with the structure tree constructed. The root concept of the structure tree *replace*, and the concept *substitute* have *REPLACE* as their representative concept. *REPLACE* has three schemas associated with it:

1. replace kth occurrence,
2. replace any occurrence, and
3. replace all occurrences.

The schema selection rule is:

```
(cond
    ((equal (representative (replacement-position I S)) ''K-th'')
      <choose schema 1>
     (equal (representative (replacement-position I S)) ''ANY'')
      <choose schema 2>
     ((equal (representative (replacement-position I S)) ''ALL'')
      <choose schema 3>)))
```

Here I and S stand for the informal specification and the structure tree, respectively. The function **replacement-position** obtains the position at which replacement occurs from S. Then the representative concept for this position is obtained (using **representative**).

K-th, *ANY*, and *ALL* denote representative concepts. For example, both "replace *every* occurrence ..." and "replace *all* occurrences ..." would have the representative concept of their replacement positions to be *ALL*. Application of the selection rule to the current problem results in schema 1 being selected. The schema retrieved is shown in Figure 2.

In the axiom scheme part several kinds of variables are present. Variables with the prefix "?var" in their names denote variables in the ultimate formal specification. Variables with "?pred" prefix stand for predicates and should be instantiated to names of predicates. Prefix "op" indicates operators and "?-" denotes input or output variables of the specification. The symbol $||$ denotes the set cardinality function, $||_{multiset}$ denotes the multiset cardinality, and $\in_{multiset}$ denotes the multiset membership.

The first line of precondition states the values of array ?-A. The second line asserts that the number of occurrences of ?-x in ?-A is at least ?-k. The postcondition locates the ?-kth occurrence of ?-x in ?-A and replaces it with ?-y.

The schema instantiation rules dictate the manner in which the axiom schemes are instantiated. Depending upon the replacement position, instantiation rule 1 sets the index limits of the array as well as the relational operators. In our example the assignments corresponding to the "first" position are made. As the occurrence condition is simply equality, rule 2 instantiates the predicate ?pred-ψ to "=". Rule 3 sets ?-k to "1" which is the cardinal number for "first". Rules 4-6 set the names of the inputs. Using these values, the axiom scheme is instantiated to:

precondition:

$$\begin{aligned}
&\forall\ \texttt{?var-i}:\ (1 \le \texttt{?var-i} \le n \Rightarrow A[\texttt{?var-i}] = b_{\texttt{?var-i}}) \\
&|\{\ \texttt{?var-i}:\ 1 \le \texttt{?var-i} \le n \wedge A[\texttt{?var-i}] = x\ \}| \ge 1
\end{aligned}$$

postcondition:

$$\begin{aligned}
\exists\ \texttt{?var-j},\ \texttt{?var-s}:&(\ \texttt{?var-j} < n) \wedge \\
\forall\ \texttt{?var-i}:\ &(1 \le \texttt{?var-i} < \texttt{?var-j}) \wedge \\
&((b_{\texttt{?var-i}} = x) \Leftrightarrow (x \in_{multiset} \texttt{?var-s})) \wedge \\
&(|\texttt{?var-s}|_{multiset} = 0) \wedge (b_{\texttt{?var-j}} = x) \wedge (A[\texttt{?var-j}] = y)
\end{aligned}$$

At this point the formal specification derived by schema-based approach is handed over to the postprocessor. The presence of the collection type *array* in the formal specification triggers the *array invariance* demon shown paraphrased in Figure 3. Execution of this demon gives the following final definition of the replace operation.

```
SCHEMA-NAME:                  Replace K-th Occurrence
SCHEMA-PRECONDITIONS:         Nil
SCHEMA-AXIOM-SCHEMES:
  precondition:
     $\forall$ ?var-i:  ( ?-end-2 $\leq$ ?var-i $\leq$ ?-end-1 $\Rightarrow$ ?-A[?var-i] = $b_{?var-i}$) $\wedge$
     $|$ { ?var-i:  ?-end-2 $\leq$ ?var-i $\leq$ ?-end-1 $\wedge$ ?-A[?var-i] = ?-x } $|$ $\geq$ ?-k
  postcondition:
     $\exists$ ?var-j, ?var-s:( ?var-j < ?-end-1) $\wedge$
             $\forall$ ?var-i:  (?-end-2 ?op-1 ?var-i ?op-2 ?var-j) $\wedge$
                        (?pred-$\psi$($b_{?var-i}$, ?-x) $\Leftrightarrow$ (?-x $\in_{multiset}$ ?var-s)) $\wedge$
                        ($|$?var-s$|_{multiset}$ = ?-k - 1) $\wedge$ (?pred-$\psi$($b_{?var-j}$, ?-x)) $\wedge$
                        (?pred-$\alpha$(A[?var-j], ?-y))
SCHEMA-INSTANTIATION-RULES:
1.    (cond ((equal (replacement-position I S) ``first'')
                (assign ?-end-1 (upper-lim (input-collection I S)))
                (assign ?-end-2 (lower-lim (input-collection I S)))
                (assign ?-op-1 ``$\leq$'')
                (assign ?-op-2 ``<''))
               ((equal (replacement-position I S) ``last'')
                (assign ?-end-1 (lower-lim (input-collection I S)))
                (assign ?-end-2 (upper-lim (input-collection I S)))
                (assign ?-op-1 ``$\geq$'')
                (assign ?-op-2 ``>''))
               ((equal (replacement-position I S) ``?k-th'')
                (assign ?-end-1 (upper-lim (input-collection I S)))
                (assign ?-end-2 (lower-lim (input-collection I S)))
                (assign ?-op-1 ``$\leq$'')
                (assign ?-op-2 ``<'')))
2.    (assign ?pred-$\psi$ (occurrence-condition I S))
3.    (assign ?-k (cardinal-number (replacement-position I S)))
4.    (assign ?-A (name (input-collection I S)))
5.    (assign ?-x (name (input-replaced-element I S)))
6.    (assign ?-y (name (input-replacing-element I S)))
```

Figure 2: Replace K-th occurrence schema

```
Demon Array Invariance (F)
/* F is the formal specification derived so far */
1. Determine the set of indices in array at which values have changed.
/* Uses the simple heuristic of finding out all relevant occurrences of
the
array in postcondition of the formal specification.
Relevant occurrences are those which are on either side of
''='' sign in a conjunct. */
2. State invariance for the complement of the set found in Step 1 and
attach to the formal specification.
```

Figure 3: Array Invariance Demon

precondition:

$\forall$?var-i: $(1 \leq$?var-i $\leq n \Rightarrow$ A[?var-i] $= b_{?var-i})$

$|\{$?var-i: $1 \leq$?var-i $\leq n \wedge$ A[?var-i] $= x \}| \geq 1$

postcondition:

$\exists$?var-j, ?var-s:(?var-j $< n) \wedge$

$\forall$?var-i: $(1 \leq$?var-i $<$?var-j$) \wedge$

$((b_{?var-i} = x) \Leftrightarrow (x \in_{multiset}$?var-s$)) \wedge$

$(|$?var-s$|_{multiset} = 0) \wedge (b_{?var-j} = x) \wedge ($A[?var-j] $= y) \wedge$

$\forall$?var-i: $((1 \leq$?var-i $\leq n) \wedge$

$($?var-i $\neq$?var-j$)) \Rightarrow ($A[?var-i] $= b_{?var-i})$

Replacing the last occurrence of some element or some k-th occurrence of an element can be defined using the same schema. Analogous schemas are associated with concepts *FIND* and *REMOVE*. The only difference is in the ?pred-α.

5 CONCLUSION

In this paper, we have described a fully implemented system, SPECIFIER, which utilizes operation abstractions to derive formal software specifications from informal descriptions. We have demonstrated that such abstractions of operations can serve as domain independent schemas. We have presented the knowledge representations for schemas, and informal specifications, and discussed an approach to organizing the knowledge base that allows efficient use of schemas.

We have not yet explored the possibility of using inter-concept relations to modify schemas. In essence, such a modification would abstract the schema further to subsume the connected concept classes. It would be interesting to analyze the effect of (automatic) representation shifts - both of schemas and of internal representation of informal specification - on the capabilities of the system. This problem is related to that of schema modification and might result in partial specifications of problems for which the current system might fail. Finally, it would be worthwhile to study the integration of a learning component with the existing system to enable automatic schema induction after repeated use of past problems by analogy.

References

[1] R. Balzer, N. Goldman, and D. Wile. Informality in program specifications. *IEEE Trans. on Software Eng*, 4(2):94–103, March 1978.

[2] M. Bidoit, M.-C. Gaudel, and A. Mauboussin. How to make algebraic specifications more understandable: an experiment with the PLUSS specification language. *Science of Computer Programming*, 12(1989):1–38, 1989.

[3] E. Dubois. A logic of action for supporting goal-oriented elaborations of requirements. In *Proceedings of Fifth International Workshop on Software Specification and Design*, pages 160–168, May 1989.

[4] Eric Dubois, Nicole Levy, and Jeanine Souquieres. Formalising restructuring operators in a specification process. *Lecture Notes in Computer Science*, (289), 1987.

[5] Eric Dubois and Axel van Lamsweerde. Making specification processes explicit. In *Proceedings of Fourth International Workshop on Software Specification and Design*, pages 169–177, April 1987.

[6] M. S. Feather. Constructing specifications by combining parallel elaborations. *IEEE Transactions on Software Engineering*, 15(2):198–208, February 1989.

[7] M.S. Feather. Detecting interference when merging specifications. In *Proceedings of Fifth International Workshop on Software Specification and Design*, pages 169–176, IEEE Computer Society Press, April 1989.

[8] S. Fickas. Automating analysis: an example. In *Proceedings of Fourth International Workshop on Software Specification and Design*, pages 58–67, IEEE Computer Society Press, April 1987.

[9] S. Fickas. *Automating the Specification Process.* Technical Report CIS-TR-87-05, Department of Computer and Information Science, University of Oregon, Eugene, Oregon 97403. Telephone (503) 686-4408, December 1987.

[10] S. Fickas and P. Nagarajan. Being suspicious: critiquing problem specifications. In *AAAI*, pages 19–24, 1988.

[11] Cordell Green, David Luckham, Robert Balzar, Thomas Cheatam, and Charles Rich. *Report on a Knowledge-Based Software Assistant.* Technical Report RADC-TR-83-195, Rome Air Development Center, Griffiss AFB, NY 13441, June 1983.

[12] W. L. Johnson. Deriving specifications from requirements. In *10th International Conference on Software Engineering*, pages 428–438, Singapore, 1988.

[13] W. L. Johnson. Overview of the knowledge-based specification assistant. In *Proceedings of the Second Annual Knowledge-Based Software Assistant Conference*, pages 48–79, Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, NY 13441-5700, January 1988.

[14] W. L. Johnson. Turning ideas into specifications. In *Proceedings of the Second Annual Knowledge-Based Software Assistant Conference*, pages 138–153, Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, NY 13441-5700, January 1988.

[15] K. Miriyala and M. T. Harandi. Analogical approach to specification derivation. In *Proceedings of Fifth International Workshop on Software Specification and Design*, pages 203–210, May 1989.

[16] K. Miriyala and M. T. Harandi. *Automatic Derivation of Formal Software Specifications from Informal Descriptions.* Technical Report UIUCDCS-R-90-1581, University of Illinois at Urbana-Champaign, 240 DCL, 1304 W. Springfield Ave., Urbana, IL 61801, April 1990.

[17] H. B. Reubenstein and R. C. Waters. The requirements apprentice: an initial scenario. In *Proceedings of Fifth International Workshop on Software Specification and Design*, pages 211–218, May 1989.

[18] C. Rich, R. C. Waters, and H. B. Reubenstein. Toward a requirements apprentice. In *Proceeding of Fourth International Workshop on Software Specification and Design*, pages 79–86, April 1987.

TOWARDS KNOWLEDGE-BASED REVERSE ENGINEERING

J. Zhang and C. Boldyreff

Department of Computer Science
Brunel University, Uxbridge
Middlesex, England
Tel: (UK) 0895 74000

ABSTRACT

Reverse engineering promises improvements in software maintenance and reuse. However, its definition, scope, problems, and potential exploitation remain unclear. It is the purpose of this paper to present a personal view of these aspects of reverse engineering with discussions of several of its fundamental problems. It is shown that reverse engineering is quite different from the reversal of forward engineering, and software understanding and software documentation are two major tasks of reverse engineering. Our research suggests computer assisted knowledge-based approaches to reverse engineering. The roles of the software engineer, computer, knowledge for reverse engineering are identified in reverse engineering. Some existing techniques in the related research areas are surveyed.

1. INTRODUCTION

Reverse engineering has become one of the most interesting research areas in software engineering. Conventional software forward engineering starts with a high level requirement of software, and produces its lower level specification, design, and code etc. Reverse engineering, however, tries to abstract or recover higher level information about software from its lower level description for software maintenance and reuse. Despite the fact that there has been much research in the related areas of reverse engineering in recent years, the appropriate definition and scope of reverse engineering remains unclear, and this obstructs further exploitation of reverse engineering.

In this paper, we present a personal view of the definition, scope, problems, and potentials of reverse engineering with discussions of several of its fundamental problems. It is concluded that reverse engineering is quite different from the reversal of forward engineering, and software understanding and software documentation are two major tasks of reverse engineering. Our research suggests computer-assisted knowledge-based approaches to reverse engineering. The roles of the software engineer, computer, knowledge for reverse engineering are identified in reverse engineering. The reverse engineering knowledge used by the engineer and that used by the computer usually have quite different forms and contents, and human-computer interaction is of

This research is supported by the Commission of the European Communities under the P1094 Practitioner project.

great importance in reverse engineering. It is essential for a computer-assisted reverse engineering system to support users with multiple views of software in order to have better use of human knowledge of reverse engineering. The uses of formal methods and code level information are of great importance in moving towards automated reverse engineering. Some existing techniques in the related research areas are surveyed.

2. WHAT IS REVERSE ENGINEERING?

The need for reverse engineering has been widely discussed and recognised [ANT87,ARN86,BIG89]. However, the definition of reverse engineering remains unclear. A typical example in the software industry is that thousands or millions of lines of undocumented or poorly documented old code need to be modified to incorporate new changes in requirements. It is infeasible to discard all old code and redevelop software by following modern development methodologies. The purpose of reverse engineering in this case *can* be to find or extract information about the software at various levels to produce good documentation for future re-engineering. The need for reverse engineering in the above example is obviously based on commercial interest. What is a proper definition for reverse engineering in general remains as a question.

A number of research activities have been recognised in the areas of reverse engineering, which together constitute the current paradigm of reverse engineering. Among these are the following major activities:

(1) Exploration of formal methods to derive specifications from code [WAR88];
(2) Restructuring of software to enhance modularitiy of designs [ARN86,BIG89];
(3) Application of compiling techniques to analyse code [AMB81,JAN81];
(4) Documentation of software from various points-of-view for maintenance [LIE78,PAR86, SWA76].

The above activities involve the extraction and representation of information at specification, design, coding and maintenance levels respectively. Simply gathering together these activities does not of itself lead us to an appropriate definition of reverse engineering. Instead it might lead to a plausible definition of reverse engineering as follows:

- *Reverse engineering* is the reversal of forward engineering in software engineering, which starts with code, and abstracts its design, specification and requirement respectively.

This definition of reverse engineering looks quite straightforward and useful at the beginning. However, when one starts to think of the definition in depth and its related research, a serious question arises, that is, can *every* necessary piece of information regarding designs, specifications, and requirements be abstracted from code? If the answer is no, the definition of reverse engineering given above will not stand up.

In order to have a better understanding of this problem, it is necessary to have a closer look at forward engineering in software engineering. The classical water fall model of software engineering suggests the following process of forward engineering:

- *Forward engineering* starts with user requirements of a software system, and through a process of successive refinement, generates a specification, design and code of the software system respectively.

Forward engineering produces much information about the software system at various levels higher than code, which, for example, may include dataflow diagrams (DFDs), entity grid charts and structure charts (LDS), entity life history matrixes (ELH), logical dialogue outlines (LDO) for dialogue design, relational data analysis documents (RDA), composite logical data design documents (CLDD), and documents of process outlines, first cut data design, first cut program design and physical design control [DOW88,PRE87].

If reverse engineering is the reversal of forward engineering, a natural question to be asked is: can the information listed above be recovered and abstracted from poorly documented or undocumented code itself? The immediate answer is likely to be *with great difficulty*. Our research and observations have led us to conclude that in many cases it is impossible to fulfil the recovery or abstraction task from code itself, even with human intervention. Our research suggests the following broad definition for reverse engineering:

- *Reverse engineering* is the post engineering of existing artifacts from a forward software engineering process, which involves improving software understanding at various levels using existing software documentation, and it results in software documentation representing this improved understanding for software maintenance.

It is no longer the goal for reverse engineering to recover all possible information regarding requirements, specifications and designs of software *unless* the existing documentation of the software is sufficient to allow this. There is also no fixed order of using and abstracting software information. One can use all the information available in the existing documentation to assist software understanding at various levels.

In the following section, we have a closer look at forward engineering and study why forward engineering is in general possible whereas the reversal of it is not. This study leads us to the broad definition of reverse engineering given above.

3. FORWARD AND REVERSE ENGINEERING

In forward engineering, we start with user requirements for a software system, and from these we generate a specification, design and realisation in code of the software system respectively. The requirements express the customer's intentions for a software system in an application domain-oriented form. A specification specifies precisely *what* a system must do to satisfy the requirements in a software developer-oriented way. A design describes language-independent details at various levels concerning *how* the functions of a specification will be realised. The realisation in code is a language-dependent implementation of a design.

We call software requirements, specifications, designs and code all *software expressions*, and denote them R, S, D, C respectively. It is clear that none of these software expressions will be self-interpreted or complete. They simply need associated knowledge for interpretation. We term such knowledge as *software expression knowledge* and denote it by K_r,K_s,K_d,K_c respectively. Thus, $R+K_r$, $S+K_s$, $D+K_d$, $C+K_c$ denotes what is needed to understand a software system completely at four different levels.

For example, at code level, all built-in routines and arithmetic operations can be called without definition, and the statements of the language have pre-defined meanings. In this case, K_c is the knowledge of the programming language. At design level, the interpretation of design documentations such as DFD, LDS etc. needs access to the knowledge K_d about the particular design representation employed. A Z or VDM specification will use a large set of pre-defined set operations or predicates. The requirement knowledge K_r of an application domain, which in most cases is informal, fuzzy, incomplete, inconsistent, plays an important role in interpreting customer's intention. That is why the requirements refinement to remove any incompleteness and inconsistencies from the requirements, by consulting the domain knowledge K_r through discussions with customers, is one of the major tasks in requirements analysis.

There is *another* kind of knowledge which is used in software development to transform one software expression to another. For example, to transform a requirement R into a specification S, a software analyst has to know not only the requirement knowledge K_r and the specification knowledge K_s, but also the knowledge to bridge R and S, which we denote K_{rs}. Similarly, we denote the knowledge used in transforming designs from specification, and code from designs as K_{sd} and K_{dc} respectively. We term all these types of bridging knowledge as *software development knowledge.*

Software development makes use of both the software expression knowledge K_r,K_s,K_d, and K_C, and the software development knowledge K_{rs}, K_{sd} and K_{dc}. The use of software development knowledge in most cases involves human creative activities. This is why almost every software development process needs human intervention at levels beyond the code. Only when the code level is reached, is the human creation no longer needed, and the compilation system able to use its embedded compiling knowledge to fulfil the rest of development tasks, i.e., transform code into object instructions. The following diagram shows a knowledge-based forward engineering module starting from a user requirement R.

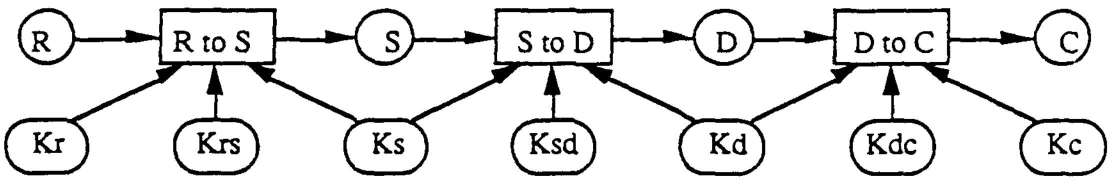


Fig.1 Knowledge-based forward engineering

The software development knowledge such as K_{rs}, K_{sd} and K_d support the top-down approach to forward engineering. Software development is a solution space pruning process. In general, for each given requirement R, there are a number of possible specifications S; for each specification S, there are a number of possible designs; and so on. The task of forward software engineering is to determine a path from the requirements R to code C in the solution space by using the development knowledge of the experienced software engineer. The following diagram shows the solution space pruning in forward engineering.

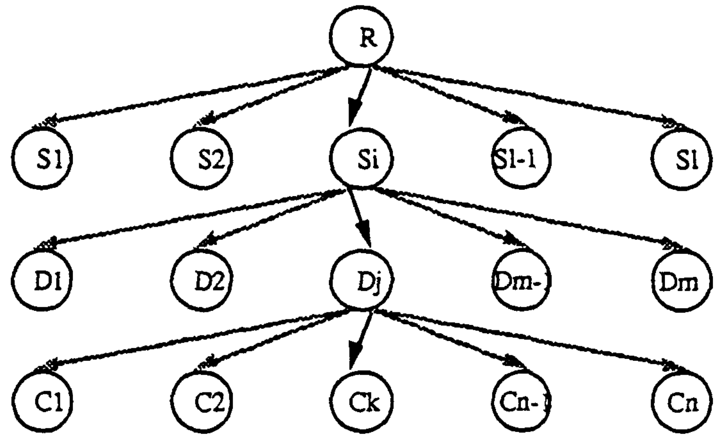


Fig. 2 Solution space pruning in forward engineering

where in the solution chosen S_i=S, D_j=D and C_k=C.

What about reverse engineering? Starting from a code C, there are in general a number of possible designs D; for each design D, there are a number of possible specification S; and so on. If we start with code C, and try to produce design D, specification S, and requirement R in reverse engineering, can we have the following tree-like solution space pruning for reverse engineering?

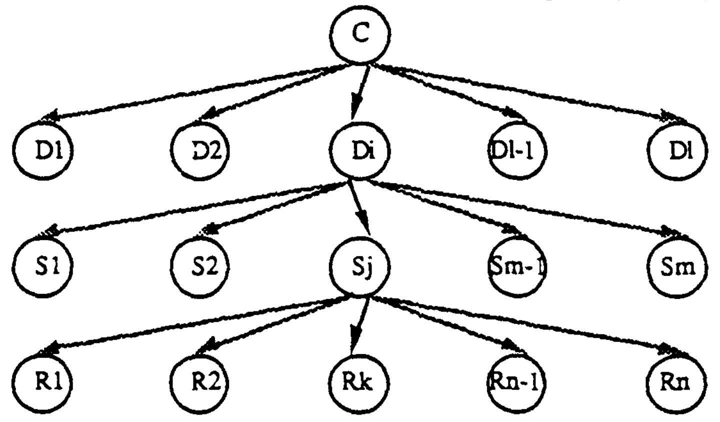


Fig. 3 Solution space pruning in reverse engineering

where D_i=D, S_j=S and R_k=R.

The answer is *negative*. There is a significant difference between forward engineering and reverse engineering. In principle, any path from a given requirement R to its resulting code C generated in forward engineering is acceptable, as long as the final software product satisfies the customers requirements. However, this is not true for reverse engineering. Reverse engineering, if possible, needs to find or recover human intentions in software coding, designs, specifications and requirements etc. from existing software documentation. It is not acceptable at all in reverse engineering to produce a path from a given code C to an arbitrary design D, specification S, requirement R.

There is a relatively strong link between code C and design D, therefore much design information concerning data structure, program structure, procedural details etc. can be easily abstracted from code C. However, there is usually a much looser link between design D and specification S. A specification S can be clearly written to specify the function of the software without any concern for efficient design. Thus, an efficient design D of S may differ substantially from the specification S in structure, which makes it extremely difficult to recover the original specification S from design D. Moreover, the link between specification S and requirement R is obviously the weakest, and it is believed that recovery of requirement R from specification S without knowing the application domain is generally impossible [TUR87].

The above discussion leads to further distinguishing between forward engineering and reverse engineering. In forward engineering, each top-down development step prunes the solution space and produces a lower level software expression. Once a step is made, there is no need for further backtracking unless some mistakes are made in this step. In reverse engineering, guess is the main characteristics of this bottom-up like process. It is much more difficult to prune the solution space in reverse engineering simply because we are not allowed to produce an arbitrary requirement R, specification S etc. from code C. To recover human intention embedded in the software code, the rigorous bottom-up reverse engineering wouldn't work; guessing and backtracking amongst the different steps are essential.

On the other hand, in general it is impossible to recover or abstract all the information concerning human intention at the levels of requirement, specification, design from code itself, unless the comments associated with code and the existing human expert knowledge about the software are sufficient. Without knowing what the application domain of a software is, it is hardly believable that one can produce a requirement for the software from its *undocumented* code. The case is the same for abstracting specification from code. Although one can transform mechanically code into its higher level representation such as Z to certain extent, the transformed representation in many cases cannot be really viewed as a specification of the code. This is because the mechanically transformed representation does not specify the software in the same way as an expert software engineer.

From the discussions above, we can easily see that reverse engineering is a quite different process from the reversal of forward engineering (which actually does not appear to be feasible or useful), that the rigorous bottom-up approach does not work in reverse engineering. It is also clear that it is not the proper purpose for reverse engineering to recover all the information about a software simply because this is not always possible. What then is the proper definition for reverse engineering?

Software engineers can reverse engineer software to certain extent, but are not limited by having to start only from code. They make use of all available information about software at various levels to increase their understanding of the software. The understanding from the software obtained in reverse engineering is then recorded and represented in a human oriented way for the future use in software maintenance. There is no guarantee that it will be possible to recover or abstract all the information about original development intention in requirement, specification etc unless the software is reasonably well documented. Besides software requirements, specification, design etc., any other views and understanding of the software can be recorded and represented in desired forms. Reverse engineering actually is the post engineering of existing software artifacts. These discussions naturally bring us to the definition of reverse engineering given in Section 2.

4. HOW TO REVERSE ENGINEER ?

The definition of reverse engineering discussed in the previous section suggests the following computer assisted reverse engineering model:

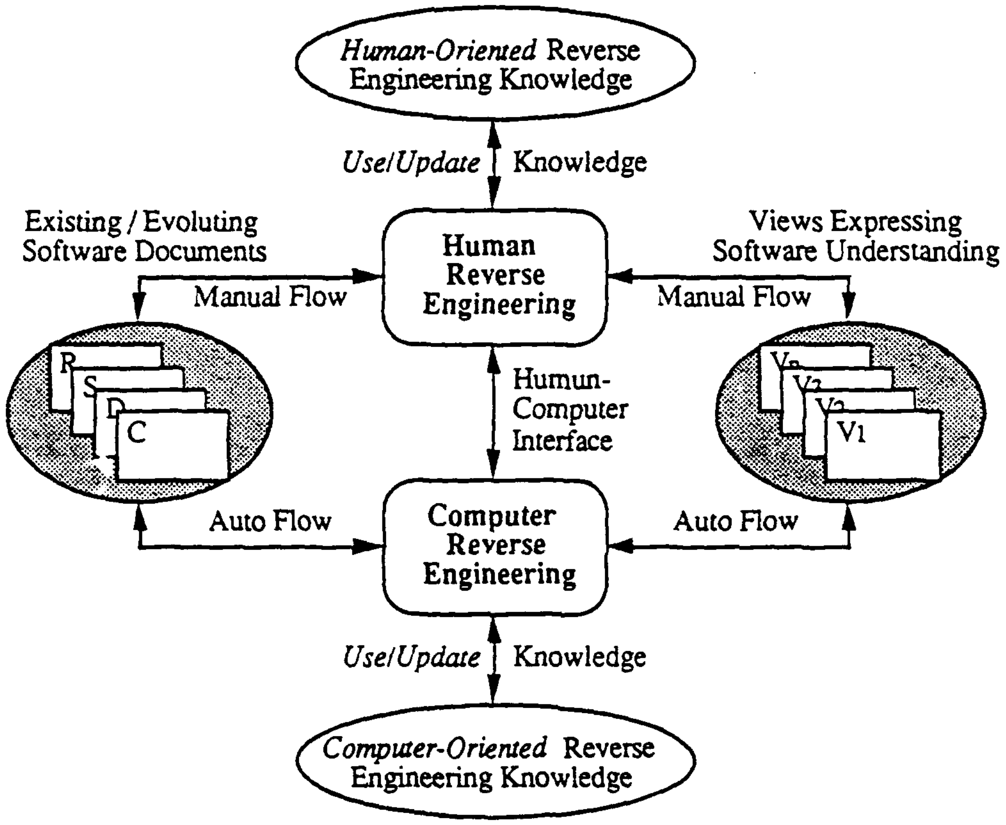

Human-Oriented Reverse Engineering Knowledge
Use/Update Knowledge
Existing / Evoluting Software Documents
Human Reverse Engineering
Views Expressing Software Understanding
Manual Flow
Manual Flow
R
S
D
C
Humun-Computer Interface
Vn
V3
V2
V1
Auto Flow
Computer Reverse Engineering
Auto Flow
Use/Update Knowledge
Computer-Oriented Reverse Engineering Knowledge

Fig. 4 Computer-assisted reverse engineering

Computer assisted reverse engineering starts with the existing documentation of a software system, typically the source code and associated documentation including comments at various levels. A computer system embedded with knowledge of automated reverse engineering techniques analyses the existing software documentation to produce new software documentation recording the captured understanding of the software. A software engineer with various additional knowledge for reverse engineering interacts with the process of automated reverse engineering, and manually analyses both the original and computer produced software documents and various views, to reach a better understanding of software with the computer. Both human and computer oriented knowledge for reverse engineering are dynamically updated and accumulated. Fig.4 illustrates an overall model of such a computer assisted reverse engineering.

Knowledge based reverse engineering involves three active parts: engineer, computer, and the *reverse engineering knowledge* base usable by either the engineer or computer for reverse engineering, which clearly includes both software expression knowledge K_r,K_s,K_d,K_c and software development knowledge K_{rs}, K_{sd}, and K_{dc}. Another kind of knowledge specially developed for reverse engineering to analyse software expressions to capture higher level understanding of software is also included in reverse engineering knowledge, which we term *software understanding knowledge*. We use K_{cd}, K_{ds}, and K_{sr} to denote the knowledge used for capturing higher level understandings concerning design D, specification S, and requirement R from code C, design D, and specification S respectively The following diagram shows a knowledge-based reverse engineering life-cycle which starts with existing software documents C_0, D_0, S_0, R_0 at four different levels, where D_0, S_0, R_0 can be empty, but usually are brief and incomplete documents.

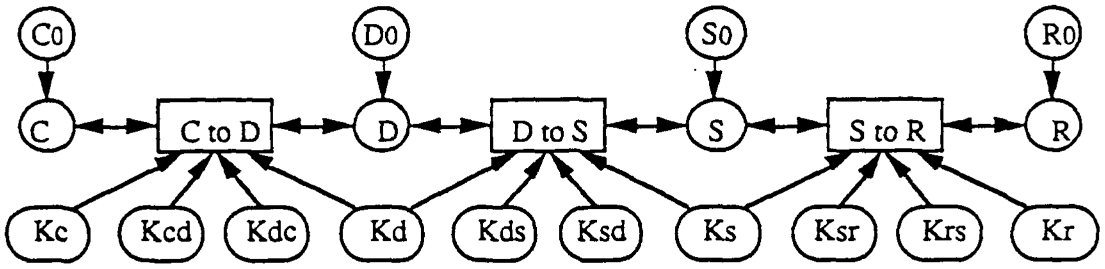


Fig. 5 Knowledge-based reverse engineering

Before reverse engineering, engineers usually roughly know *something* about the application domain, system functions, and possible design choices etc, or such information can be found in software documents D_0, S_0, R_0. This important information can be effectively used to invoke relevant reverse engineering knowledge (in fact, pruning the knowledge space), therefore enabling effective pruning of solution space for reverse engineering. One of the authors had a hard experience reading the assemble code of an assembler to understand every detail of the code, which was a typical example of reverse engineering. Before tackling the code, it was already known that it was an assembler of the language itself. After studying the existing techniques of assemblers from a text book and other references, he understood how an ordinary assembler might work. Bearing all this knowledge in mind, he finally succeeded in working out every detail of the assembler. Without knowing that the code was an assembler, the possibility that one could reverse engineer an assembler from its code is slight.

The above example has also shown that the use of software development knowledge such as K_{rs}, K_{sd}, and K_{dc} takes an important role in reverse engineering. In general, knowing the application domain of software, K_r and K_{rs} could be used to guess the application software at both the requirements and specification levels. If such guesses finally match the lower level software expressions, then the whole reverse engineering process succeeds. On the other hand, software understanding knowledge such as K_{rs}, K_{sd}, and K_{dc} assists bottom-up guesses in reverse engineering from lower levels to higher levels. Reverse engineering is basically a two-way software pattern matching process between both bottom-up guesses for software understanding and top-down guesses for software development.

Both human and computer-oriented reverse engineering knowledge belongs to human knowledge, and is obtained mainly from human creative activities (possibly with some computer assistance). Human-oriented knowledge of reverse engineering contains all human knowledge of reverse engineering human readable forms. However, computer-oriented knowledge of reverse engineering consists of only the knowledge which has been well understood and can be formally expressed in a way usable by computer in reverse engineering. This is especially true for the knowledge concerning application domains at the requirements level. Using only computer-oriented knowledge in reverse engineering clearly limits the process. The need for the use of human-oriented reverse engineering knowledge by software engineers is obvious in reverse engineering.

One widely recognised problem in reverse engineering is the difficulty of recovering requirements from their lower level software expressions. As we have discussed before, the link between requirements and specifications is the weakest among all the links between requirement R, specification S, design D and code C, which partly explains the reason for such a difficulty. Moreover, the major domain knowledge K_r concerning software requirements itself cannot be obtained through reverse engineering (which actually belongs to the tasks of knowledge engineering and domain analysis), and needs to be provided either by computerised knowledge bases or by domain analysts engineers. Due to the usual huge size and informality of domain knowledge, it is infeasible to build a complete knowledge base beforehand. Incremental evolution of certain kinds of computerised domain knowledge bases, as well as the use of human resource of domain knowledge are essential for reverse engineering.

It is clear that software understanding and representing the understanding of software are two major tasks in reverse engineering. Because of the need for the expertise of the software engineer in reverse engineering, it is very important to have CASE tools to assist the engineers in visualising the understanding of the software captured by the computer. Until the human users have better knowledge of software, their expertise in reverse engineering cannot be fully exploited. On the other hand, one goal of reverse engineering is to represent software understanding for future software maintenance; therefore, it is vital to represent software understanding in a human-oriented way.

It is not necessary to present a complete view of software, such as a complete formal mathematics specification of a software. Any partial views of a software at desired angles may be useful [REI85]. For example, the existing compiling techniques such as control flow analysis and data flow analysis can be used to analyse code to produce visual diagrams to help the engineers to reverse engineer software. Data base technology can be used to store views, and support user queries for information about software, such as file usage, cross-reference of variables [CHE86,FOS87]. Software browsers can be used to view the software in a top-down style. Dynamical testing and other software quality measurement techniques such as time and space complexity measurement techniques also can contribute to reverse engineering. Any other software documentation standards and techniques that are useful for enhancing human understanding of software can assist in its reverse engineering. Advanced user-computer interfaces and hypertext system [CON87] with graphics and windows are essential to reach the desired performance [MOR13].

On the other hand, although the human involvement is inevitable in reverse engineering, it is the main goal for reverse engineering to reduce the human intervention as much as possible by developing computer assisted tools and automated self-applied approaches with enriched computer-oriented reverse engineering knowledge. The main difficulty of automated reverse engineering is that most knowledge of software expressions and software development exists informally, and can only be applied semi-automatically with human intervention in software engineering, both forward and reverse. Formal methods have promising potential in this aspect. Other techniques such as those from Cognitive Science aiming to reduce human effort with better solutions to the problems are also expected to contribute towards increased automation of reverse engineering.

Capturing human intentions from software code or other software documentation is one of the most difficult tasks in reverse engineering. Although it is possible to mechanically transform software code into a Z-like specification, or an intermediate program written in a wide spectrum language, such transformed Z-like specification or intermediate program is usually quite different from what an expert engineer might produce in the same language. Much effort needs to be taken in the later stages to promote the mechanically transformed specification to match the human intentions. Current mechanical transformation techniques tend to produce unreadable specifications. Potentially knowledge-based methods could have an important role in this area of research.

6. CONCLUSIONS AND REMARKS

In this paper, we have presented a personal view of the definition, scope, problems, and potentials of reverse engineering. It has been shown that reverse engineering is quite different from the reversal of forward engineering, and software *understanding* and *software documentation* are two major tasks of reverse engineering. Our research suggests the computer-assisted knowledge-based approaches for reverse engineering. The roles of the software engineer and computer, and type of knowledge required for reverse engineering are identified. It is believed that human involvement in reverse engineering is essential, and the computer support should be given for users to have multiple views of software for better use of human knowledge of reverse engineering. The use of formal methods and code level information is also of great importance in obtaining automated reverse engineering.

This paper has proposed a framework and a process model for reverse engineering. The work presented here directly came from the research on software reuse in the Practitioner Project. The project aims at reusing software components at conceptual levels, and reverse engineering has been investigated to support identification of reusable software components. A number of related case studies are being conducted to explore transformational methods for code analysis, design discovery, and specification abstraction etc. In particular, incremental knowledge-based methods have been adapted to incorporate transformational approaches to realise reverse engineering at the requirements level.

REFERENCES

[AMB87] Ambras, J. and V.O'DAy, "Microscope: A program analysis system," *Proceedings of Hawaii International Conference on System Sciences-20* (January 1987), pp. 71-81.

[ANT87] Antonini, P. *et al*, "Maintenance and Reverse Engineering: Low-level Design Documents Production and Improvement", *Proceedings of Conference on Software Maintenance*, Austin, Texas, pp. 91-100, 1987.

[ARN86] Arnold, S., *Tutorial on Software Restructuring*, IEEE Computer Society, 1986.

[BIG89] Biggerstaff, J., "Design Recovery for Maintenance and Reuse," *IEEE Computer*, Vol. 22, No. 7, pp. 36-49, July 1989.

[BOL89a] Boldyreff, C., P. Hall, and J. Zhang, "Reusability: The Practitioner Approach", Position paper printed in: *Workshop Reuse: Research in Progress*, Delft University of Technology, November 1989.

[BOL89b] Boldyreff, C., and J. Zhang, "From Recursion Extracion To Automated Commenting - A transformational approach towards reverse engineering of software to support reusability", In: *Unicom Workshop: Reuse, Maintenance and Reverse Engineering of Software*, November 1989.

[CHE86] Chen, Y. and C. V. Ramamoorthy, "The C information abstractor," *COMPSAC 86*, Chicago (October 1986), pp. 291-298.

[COL85] Collofello, J. S., and J. W. Blaylock, "Syntactic information useful for software maintenance," *National Computer Conference 85*, Chicago (July 1985), pp. 547-553.

[CON87] Conklin, J., "Hypertext," *IEEE Computer* 20, No. 9, pp.17-41 (September 1987).

[DOW88] Downs, E., and P. Clare, I. Coe, *Structured Systems Analysis and Design Method*, Prentice Hall, 1988.

[FOS87] Foster, R., and M. Munro, "A Documentation Method Based on Cross-referencing", *Proceedings Conference on Software Maintenance 1987*, pp. 181-185, 1987.

[JAN81] Jandrasics, C., "SOFTDOC - A System for Automated Software Analysis and Documentation," *Proceedings of ACM Workshop on Software Quality Assurance*, April, 1981.

[JEN75] Jensen, K., and N. Wirth, *PASCAL User Manual and Report*, Springer-Verlag, Second Edition, 1975.

[LIE78] Lientz, B. P.,, E. B. Swanson, and G.E. Tomkins, "Characteristics of application software maintenance," *Communication of the ACM* 21, No. 6, 466-471 (June 1978).

[MOR86] Moriconi, M., and D.F. Hare, "The PegaSys System: Pictures as formal documentation of large," *ACM Transactions on Programming Languages and Systems* 8, No. 4, pp-524-546 (October 1986).

[PAR86] Parikh, G., *Handbook of Software Maintenance*, John Wiley & Sons. Inc., New York (1986), p.14.

[PAR72] Parnas,D. L., "On Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol 15, No 12, pp 1053-1058, December 1972.

[PRE87] Pressman, R. S., *Software Engineering*, McGraw-Hill, Computer Science Series, Second Edition, 1987.

[REI85] Reiss, S., "PECAN: Program development systems that support multiple views," *IEEE Transactions on Software Engineering* SE-11, No. 3, pp. 30-41 (March 1985).

[SNE89] Sneed, H. M., and G. Jandrasics, "Inverse Transformation from Code to Specification," *Proceedings of Software Tools' 89*, Blenhiem Online, London, 1989.

[STA84] Standish, T. A., "An essay on software reuse," *IEEE Transaction on Software Engineering* SE-10, No. 5, pp. 404-497 (September 1984).

[STE86] Sterling, L., and E. Shapiro, *The art of Prolog*, The MIT Press, 1986.

[TUR87] Turski, M. W., and T. S. E. Maibaum, *The Specification of Computer Programs*, Addison-Wesley, 1987.

[SWA76] Swanson, E. B., "The dimension of maintenance," *Proceedings of the Second International Conference on Software Engineering*, San Franciso (October 1976), pp. 492-497.

[WAR88] Ward, M., *Transforming a Program into a Specification*, Computer Science Technical Report 88/1, School of Engineering and Applied Science, University of Durham, January 1988.

Verifying Expert Systems using Conceptual Models[1]

Alun D. Preece
Department of Computer Science, Concordia University
1455 de Maisonneuve Boulevard West
Montreal, Canada H3G 1M8

Abstract

Building expert systems is a process of modeling knowledge at two distinct levels: a conceptual model describing the knowledge base in abstract terms, and a design model describing how to implement the knowledge base. Building these distinct models has important ramifications for knowledge acquisition, system design, maintenance, and evaluation. In particular, it supports verification of the internal consistency, conciseness and completeness of the conceptual model, and verification of the implementation with respect to the conceptual model. Our discussion of these issues is based on a pilot study.

Introduction: Modeling Knowledge

The development of expert systems can be viewed as a modeling activity [9]. Two distinct and very important models can be identified in this activity: a *conceptual model* which describes the system as a body of knowledge, and a *design model* which describes the system as an artifact.

Conceptual models are abstract descriptions of the entities, relations, and tasks that the system 'knows about', represented at a level which is independent of how these knowledge components will be represented and automated in an implementation of the system [9]. Examples of such abstractions are the generic tasks of Chandrasekaran [2] and the heuristic classification model of Clancey [3].

Design models are descriptions of how to implement the knowledge structures at the conceptual level, and contain descriptions of the knowledge representations and the inference engine. They also describe the 'non-knowledge' components of the system, such as the user interface and other purely procedural features.

Making this distinction between epistemological and implementation issues in expert system development carries a number of significant advantages. The two models serve

[1]This work has been funded by the SERC in the U.K., and by Bell Canada.

as separate specifications of the knowledge base and implementation decisions. This assists the design process, in particular by easing the task of choosing representations and reasoning procedures because the conceptual structure of the knowledge can be understood better [9]. Knowledge base maintenance is also aided because the 'real' knowledge is not obscured by implementation constraints [8]. In this paper, we focus on how expert system evaluation, in particular knowledge base *verification*, is made more effective by using the separate models.

Verifying Knowledge

Verification (of expert systems) is the process of determining the internal consistency and completeness of knowledge bases [6]. It is a distinct activity from *validation*, which establishes if a system actually does the job it was designed for. Both are needed because a system can be both internally consistent and complete and still not meet all its objectives; moreover, validation testing will in general be unable to detect all inconsistencies and incompleteness in a knowledge base [6].

In general, verification of knowledge base consistency, completeness and conciseness is possible only when the knowledge is separated from implementation details, which is why conceptual models play an important role. In this paper we focus on two aspects of verification: verifying that a *knowledge base* is consistent, complete and concise at the conceptual level, and verifying that an *implementation* is consistent, complete and concise with respect to the conceptual model.

Consistency and Completeness Checking

Rule-based expert systems are related, albeit tenuously, to formal logic [1]. In particular, *if-then* production rules relate to logical implications, and rule-based expert system inference methods relate to logical rules of inference (most commonly the resolution principle [1]). This relationship has inspired the development of automated checking methods for rule bases which have identifiable logical semantics [6]. Such methods are used to determine the internal self-consistency and completeness of rule bases, by syntactic inspection and manipulation of the rules as logical expressions. The *conciseness* of the knowledge is also checked, by searching for rules or chains of inference which are *redundant* [4].

Note that the use of the terms *consistency* and *completeness* in this context differs from their use in formal logic. Here, a knowledge base is consistent if and only if there is no way that a contradiction can be asserted from valid input data [4]. A knowledge base is strictly complete if and only if it can cope with all possible situations that can arise in its domain. This will be very difficult to determine for most practical applications. The term *completeness checking*, however, refers to a syntactic method for locating logical cases for which no conclusions can be inferred by the rule base—possibly indicating missing rules [6]. Completeness checking as such addresses only part of the problem of knowledge base completeness—semantic completeness is not addressed by this syntactic checking method.

The consistency (and redundancy) check in its most basic form only detects conflicts (or redundancies) between rule pairs. For example, the rules $a \wedge b \rightarrow c$ and $a \wedge b \rightarrow \neg c$ are in conflict, while in the rule pair $a \wedge b \rightarrow c$ and $a \rightarrow c$ the former is redundant (because it is subsumed by the latter). More powerful methods detect conflicts and redundancies arising over chains of inference [4,6], as illustrated by the following example, in which the inference chain of Rules 1 and 2 is in conflict with Rule 3:

$$a \rightarrow b \tag{1}$$
$$b \rightarrow c \tag{2}$$
$$a \rightarrow \neg c \tag{3}$$

The completeness check, in its most sophisticated form [6], examines the rule base as a whole, looking for combinations of input data that are not covered by the rules. In the small rule base below (in which g is the goal, and data items a and b can have only the values *true* and *false*) the combination $a =$*false* and $b =$*true* is the only combination not covered by the rules.

$$\begin{aligned} a = \mathit{true} &\rightarrow g \\ a = \mathit{false} \wedge b = \mathit{false} &\rightarrow \neg g \end{aligned}$$

Note that use of these methods is dependent upon knowing the rules by which inferences will be drawn from the knowledge. Therefore, the basic inference rules must be described at whichever modeling level we choose to verify using the above procedures. Furthermore, use of these verification procedures requires that the knowledge representation and inference rules be chosen to conform with formal logic [1].

Verification Between Modeling Levels

We need to verify that the knowledge base of the implemented system is consistent, complete and concise with respect to the conceptual model. In some cases, it will be possible to achieve this by automatic transformation of conceptual model to implementation. This means that, if the conceptual model can be shown to be logically consistent, complete and concise, then we can be assured that the implementation is as well.

The remainder of this paper investigates the use of distinct modeling levels, and the application of the verification methods to a pilot project.

The Pilot Study

Health care is not confined to hospitals. When a patient returns home after stay in hospital, some sort of aftercare is usually required: from the arrangement of a check-up hospital appointment to the provision of a complex set of medical support groups and equipment. Organizing such aftercare requirements is called *discharge planning.*

The aim of a discharge plan is to enable patients to live as independently as possible in the community. Discharge planning requires expertise from a combination of domains and disciplines, including medical, paramedical, nursing, social work and mental health knowledge. Consequently, it is a complex activity, particularly for geriatric patients.

For this reason, it was proposed that an expert system be developed to support discharge planning. Following a period of initial study of the domain, an outline structure for the full scale system, called 'DISPLAN', was specified[2]. This structure is shown in Figure 1. On admission to hospital, the patient is *assessed* as a first step in identifying the *needs* which will have to be provided for on discharge. These needs are the basis for establishing a set of *support* items, which constitutes the aftercare requirements. Finally, a discharge *plan* is created so that all of the support items can be arranged for the patient's discharge. It is likely that changes in the patient's circumstances will result in modifications to the sets of needs, supports and discharge plans. Throughout the process, the knowledge base is applied and the patient database is both accessed and updated.

The following section describes the creation of a conceptual model for the discharge planning knowledge base.

Conceptual Model Development

There are two main components to a conceptual model: *static knowledge*, describing entities and relations in the domain, and *dynamic knowledge*, describing inference rules, tasks and procedures in the domain. Early interviews and a literature study revealed four

[2]A more detailed discussion of the DISPLAN project appears in [5]

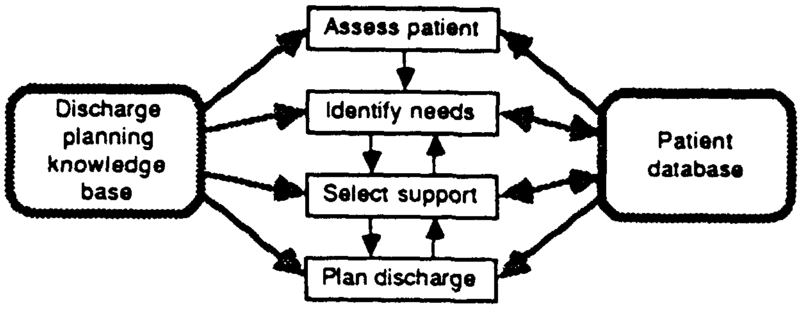


Figure 1: The Structure of the DISPLAN System.

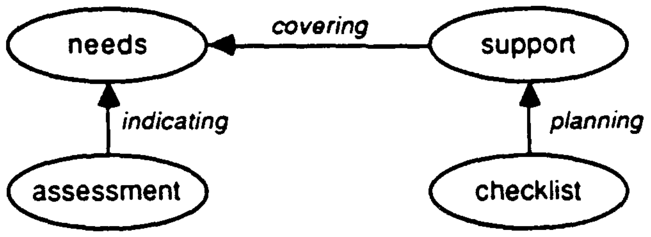


Figure 2: Entities and Relations in the DISPLAN domain.

main sets of entities and three main relations in the discharge planning *domain*, shown in Figure 2. Examples of specific instances of these entities and relationships appear below (the conceptual model used a structured English representation, as in these examples):

mobility=needs-equipment indicates *mobility-equipment-needed*

walking-frame covers *mobility-equipment-needed*

supply-walking-frame plans *walking-frame*

The dynamic knowledge associated with each of the three relations was studied. The *indicating* relation was characterized by definitional and qualitative inference rules, as described in Clancey's conceptual model of heuristic classification [3]. Similarly, the *planning* relation was a refinement inference rule from the same model. The difficult part was the *covering* relation. The first model of this was based on a mathematical set-covering model [7], in which the set of support was defined as a *minimal cover* of the set of needs (the minimization criterion was the cost of each support item). However, for a large set of needs and supports this model suffers combinatorial explosion [7].

To overcome this problem, protocol analysis was used to identify heuristics used by the discharge planning team in assigning sets of support. An example heuristic is that certain critical support services are assigned first, regardless of cost, and these early assignments constrain later choices. The overall task strategy applied was as follows:

```
Establish initial set of support:
    assign services to critical needs;
    assign equipment where appropriate;
Establish refined set of support:
    assign family carers where appropriate (and willing);
    assign services to cover remaining needs;
```

Use of this strategy resulted in more complex relationship descriptions in the conceptual model, since they had to take into account the contexts in which each relationship would hold. These constraints were expressed in the form of conditional statements, illustrated by the following example:

preparing-meals=unable indicates *preparing-meals-need*

carer-support covers *preparing-meals-need*
    if *carer-is-willing-to-prepare-meals*

In this form, the full-scale conceptual model for DISPLAN consisted of just over 3000 description statements (including data objects and their values, relationships, implications, and task descriptions). It was therefore natural that automated assistance be sought in checking this model for inconsistencies and incompleteness.

Conceptual Model Verification

As we explained earlier, the use of logic-based verification methods required the use of logic-based representation and inference methods at the conceptual level. Therefore, the DISPLAN conceptual model was transformed from the structured English representation to a proper subset of *predicate calculus* (using the Prolog language), and *backward chaining* was chosen as the basic inference method (since it is similar to the resolution method used to execute Prolog programs). The relationship statements were transformed so that the basic *indicating, covering* and *planning* inference steps would be effected by backward-chaining.

The transformation of the conceptual model was done mechanically by a Prolog program, to ensure that the transformed model was entirely equivalent to the original version. This transformation procedure was essentially a straightforward one-to-one symbolic translation[3]; the transformed versions of the example statements from the previous section appear below:

need(mobility-equipment) ← *has-value(mobility, needs-equipment).*

covers(walking-frame, mobility-equipment).

plans(supply-walking-frame, walking-frame).

need(preparing-meals) ← *has-value(preparing-meals, unable).*

covers(carer-support, preparing-meals) ← *carer-is-willing-to-prepare-meals.*

This knowledge could then be checked using the completeness, consistency and redundancy checking procedures described earlier. From the 3000 knowledge statements, approximately 100 errors were detected, indicating the value of these checking methods (these results are fully described in [6]).

Design Model Development

Developing the design model involved two main tasks. The first was designing the high-level control for the system (for scheduling the main tasks: patient assessment, needs identification, support assignment and checklist planning tasks—see Figure 1), and designing the expert system interfaces (between system and users, and between system and patient database). The second was choosing the knowledge representation and inference engine for implementing the discharge planning (static and task) knowledge of the conceptual model.

Designing DISPLAN User Interface and Control

These components of the system had no counterparts at the conceptual level, since they were entirely procedural—no different, in principle, from similar components of conventional programs. However, the complexity of the DISPLAN system required that these components be explored by prototyping (as opposed to being fully specifiable in advance), so a flexible design tool was required. For more details on these aspects of DISPLAN design, see [5].

[3]The semantics of the original structured English representation were equivalent to those of the predicate calculus subset. The translation merely facilitated easier application of the verification procedures.

The *Crystal* rule-based fourth generation language (4GL)[4] was chosen, because of its good development environment and user interface design facilities. The user interface, patient database interface, and control procedures were developed using this tool, incorporating links between these components and the discharge planning knowledge base. These components were not verifiable in the logical sense, since they lacked the clear semantics of the knowledge-based components.

Designing Knowledge Representation and Inference Engine

Experience with the Crystal tool for designing the user interface and control components of DISPLAN suggested that the rule-based representation of Crystal would be suitable for implementing the conceptual knowledge base. In effect, Crystal would be used as an expert system shell, providing the knowledge representation and inference engine. This would be especially convenient, given that the control and interface components had already been written in Crystal. The only drawback was that the tool was basically a forward chaining rule interpreter, so implementation of the static and task knowledge would require some transformations to the rules. If this transformation was done manually, there would be no assurance that the implemented system would be equivalent to the conceptual model, so it was desirable that the transformation be done automatically.

The main aspects of the transformation were: 'mimicking' the backward chaining strategy using forward chaining, and transforming the set-based data structures of the conceptual model to array-based data structures of Crystal. Both of these transformations were automatable, but the resulting Crystal code was not as comprehensible as the original predicate calculus—as shown in Figure 3 (in which the fact `'need for walking frame has been assessed'` causes a value to be sought for the data item `needs$[walking_frame]`, thereby mimicking backward chaining). This was not a problem, however, since future maintenance could be done at the higher level, and the transformation re-applied.

The example in Figure 3 also demonstrates the difficulty in verification mentioned earlier: implementation issues often obscure epistemological issues in expert system prototypes, which is why verification must be done at a higher level than that of the implementation.

Of course, such verification only ensures the *logical* consistency and completeness of the system; the implementation must still be run on test cases to *validate* the knowledge, but that topic is beyond the scope of the current paper—see [5] for a discussion of validation performed on the DISPLAN system.

[4]Developed by Intelligent Environments Ltd, Richmond, Surrey, England.

Desired action:

(1) Check whether patient needs a walking frame;
(2) If so, assign a walking frame to the set of support.

Predicate calculus (backward-chaining):

```
     includes(equipment_support, walking_frame) :-
     includes(patient_needs, walking_frame).
```

Crystal (forward-chaining):

```
     'walking frame equipment has been assigned' IF
          'need for walking frame has been assessed' AND
          needs$[walking_frame]='true' AND
          equip_support$[walking_frame]:='true'
```

Figure 3: Example equivalent predicate calculus and Crystal expressions.

Conclusions

In this paper we have presented an expert system development strategy that uses distinct conceptual model and design model stages, and we have shown that this strategy enables effective verification of the knowledge content of the system. This verification occurs in two steps: we verify the completeness, conciseness and consistency of the knowledge base at the conceptual level, and then automatically transform the conceptual model to the implementation, ensuring that the implementation is complete, concise and consistent with respect to the conceptual model.

Verification of the conceptual model requires that the semantics of the model be known. In our case, the choice of a rule-based representation and resolution-based inference method was made in order to take advantage of logic-inspired verification techniques. However, promising work in formalizing other representation and inference models [3,2,9] suggest that this development strategy can be applied to a wider range of expert system models.

Our solution to verifying that the implementation was complete, concise and consistent with respect to the conceptual model was to employ a mechanical transformation procedure. If no such procedure exists, then the implementation must be independently verified, which means that the implementation representation and inference engine must support such an approach, as discussed above.

Acknowledgements

I thank my colleagues Aida Batarekh, Anne Bennett, Peter Grogono, Rajjan Shinghal and Ching Suen for their comments and contributions, and Laurie Moseley of University College Swansea, U.K., for his contributions to the DISPLAN project.

References

[1] A. Bundy. How to improve the reliability of expert systems. In D. S. Moralee, editor, *Research and Development in Expert Systems IV.*, pages 3-17, Cambridge U. Press, 1988.

[2] T. Bylander and B. Chandrasekaran. Generic tasks for knowledge-based reasoning: the "right" level of abstraction for knowledge acquisition. *International Journal of Man-Machine Studies (UK)*, 26(2):231-243, February 1987.

[3] W. J. Clancey. Heuristic classification. *Artificial Intelligence (Netherlands)*, 27:289-350, 1985.

[4] A. Ginsberg. Knowledge-base reduction: a new approach to checking knowledge bases for inconsistency & redundancy. In *Proc. 7th National Conference on Artificial Intelligence (AAAI 88)* (St. Paul MN), Volume 2, pages 585-589, August 1988.

[5] A. D. Preece. DISPLAN: designing a usable medical expert system. In D. Berry and A. Hart, editors, *Expert Systems: Human Issues*, pages 25-47, Chapman and Hall, London, 1990.

[6] A. D. Preece. Verification of rule-based expert systems in wide domains. In N. Shadbolt, editor, *Research and Development in Expert Systems VI.*, pages 66-77, Cambridge U. Press, 1989.

[7] J. A. Reggia, D. S. Nau, and P. Y. Wang. Diagnostic expert systems based on a set-covering model. *International Journal of Man-machine studies*, 19:437-460, 1983.

[8] A. van de Brug, J. Bachant, and J. McDermott. The taming of R1. *IEEE Expert (US)*, 1(3):33-39, Fall 1986.

[9] B. Wielinga and G. Schreiber. Future directions in knowledge acquisition. In N. Shadbolt, editor, *Research and Development in Expert Systems VI.*, pages 288-301, Cambridge U. Press, 1989.

Detecting Interference in Knowledge Base Systems [*]

James G. McGuire[†] Randy Stiles[‡]

Lockheed Artificial Intelligence Center

Abstract

When a new rule is added to a knowledge base, it is difficult to predict how it will interact with the existing rules. Some of these interactions may be undesirable. The problem of interfering subgoals, which first appeared as *Sussman's Anomaly* in linear planners such as STRIPS, is a problem which can appear in any knowledge based system. The presence of interference may make rule antecedents unsatisfiable in all consistent knowledge base states. This paper describes various manifestations of interference and outlines the implementation of interference detection as part of the DARPA Expert system Validation Associate (DEVA). DEVA was built with the philosophy that automated tools capable of uncovering anomalous rule interactions can help insure the integrity of a knowledge based system over its entire life cycle.

1 Introduction

The growing use of knowledge based systems (KBS's) in industry requires the development of appropriate methods and tools to evaluate a system's correctness, consistency, and completeness. Furthermore, as these systems become operational, an increasing number of knowledge engineers will be involved in their development and maintenance. Hence, insuring the integrity of a particular KBS over its entire life cycle makes the need for automated validation even more critical.

[*]This work was performed under DARPA contract F30602-88-C-0130, monitored by Rome Air Development Center/COES.

[†]Orgn 96-20, Bldg 259, 3251 Hanover St., Palo Alto, CA 94304-1191, mcguire@laic.lockheed.com, (415)354-5586

[‡]Orgn 96-20, Bldg 259, 3251 Hanover St., Palo Alto, CA 94304-1191, stiles@laic.lockheed.com, (415)354-5256

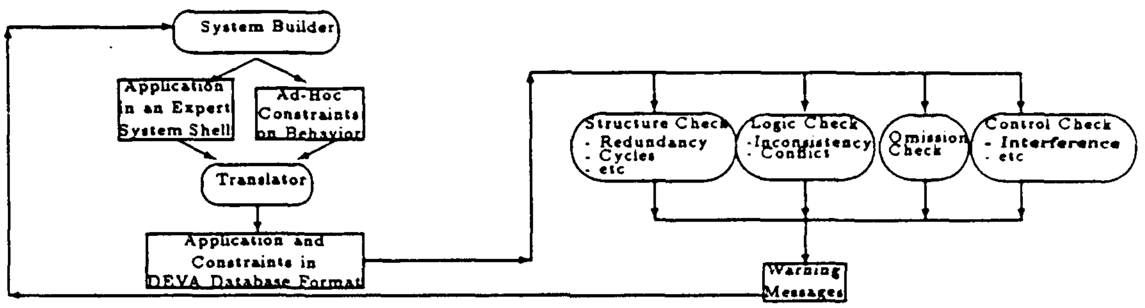


Figure 1: Subset of DEVA Functionality

The DEVA system [14,15,11] is being developed jointly by Lockheed's Software Technology Center and Artificial Intelligence Center using Quintus Prolog (See Figure 1 for an architecture overview). DEVA improves the development process by finding mistakes and omissions in the knowledge base, by proposing knowledge base extensions and modifications, and by showing the impact of changes to the knowledge base. DEVA is based on a general-purpose architecture which checks applications written in expert systems shells such as ART, KEE, CLIPS, and OPS5. A translator maps an application from the shell language to a general and declarative meta-language represented by Prolog data-structures. To tailor the checking to a specific application, meta-knowledge containing ad-hoc constraints on behavior can be input to DEVA. A suite of "checkers" is employed to statically analyze the translation for specific types of anomalies, warning the system developer/maintainer of any violations.

When a new rule is added to a knowledge base, it can be difficult to predict how it will interact with the existing rules in the system. An anomaly can result from of an unforeseen pattern of interaction between the rules. For example, consider the following rule-set:
$A \rightarrow E$
$B \rightarrow F$
$E \wedge F \rightarrow C$
Adding the rule $A \wedge B \rightarrow \neg C$ to this rule set will cause an instance of **rule-inconsistency**. Whenever the newly added rule is satisfiable, the new knowledge base will be able to derive the inconsistency $C \wedge \neg C$. Techniques for uncovering rule-inconsistency in quasi first-order logic rule-bases are presented in [11,8]. Earlier discussions dealing with propositional logic rule-bases can be found in [13,7]

The purpose of this paper is to present methods for detecting the related problem of **interference**. If a rule $LHS \rightarrow RHS$ exists in a rule base, it is expected that the knowledge base will be capable of deriving a state where the antecedents (LHS) of the rule are jointly satisfiable. If it can be shown that satisfaction of a rule's antecedents would violate semantic integrity constraints, then the rule contains unsatisfiable preconditions making the rule useless. Detection of unsatisfiable rules can be reduced to the detection of subgoal interference,

meaning the requirements to satisfy one subgoal precludes satisfaction of another subgoal. For example, consider the following rule-set:
$A \wedge B \rightarrow Goal$
$\neg F \rightarrow A$
$F \rightarrow B$
The first rule in this example will be unsatisfiable, since the subgoals $A \wedge B$ interfere with each other. To jointly satisfy both, we must prove the inconsistency $F \wedge \neg F$.

We now briefly discuss the relationship between interference and rule inconsistency. To detect rule-inconsistency, one attempts to show that there could exist fact-base scenarios (not violating any integrity constraints) capable of deriving inconsistencies. On the other hand, interference occurs when an inconsistent knowledge-base state (one which violates integrity constraints) is required to derive a rule's antecedents. Thus the two anomalies are the converse of each other. In one case we attempt to prove inconsistencies from seemingly valid support. In the other case we attempt to prove that inconsistent support is required to prove a valid goal (i.e. the antecedents of a given rule)

The remainder of this paper describes the interference module contained in the control checking component of the DEVA system. First we briefly describe the relationship between interfering subgoals in rule antecedents and Sussman's anomaly in planning systems. We then present some definitions to more rigorously describe interference detection. Next we describe the interference problem in the context of monotonic rule-bases, presenting two search strategies for performing detection. Many real-world knowledge bases are nonmonotonic, supporting .egation-by-failure and containing rules with side-effects. This further complicates interierence detection, since the operational semantics of rules possessing side-effects need to be considered. For that reason, we also include a section on interference detection over rule-bases supporting non-monotonic features.

2 Interference in Planning Systems:

The Artificial Intelligence planning community has long known of the pitfalls of interference. Interference during the satisfaction of a pair of conjunctive goals was first identified in the Blocks World and labelled as *Sussman's Anomaly* [2,17]. For the following example of Sussman's Anomaly, in Figure 2 consider a world where only one block at a time can be moved, and any block must first have a clear top before it can be moved.

The intended state is encoded as the conjunctive goal $on(A, B) \wedge on(B, C)$. The planner tries satisfying the first conjunct, by moving A onto B. A's top must be clear, so C is moved onto the table, then since B is already clear, A is put onto B. Bravo, the first goal is done. Now the planner evaluates the second goal, in which B must be on C. So it clears A from B, and now that B is clear, it puts B on top of C, and the second goal is satisfied. Of course, the first goal was undone, so the intended KB state is not the same as the final KB state.

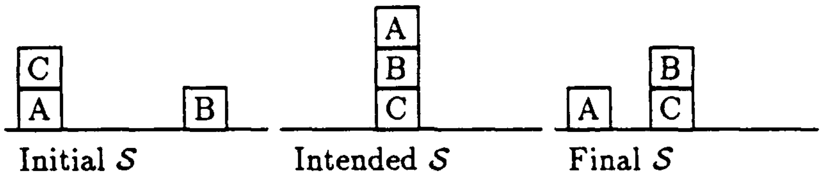


Figure 2: Sussman's Anomaly.

For interference in planning, the basic idea is that carrying out one part of the plan undoes the results of previous parts of the plan, through some use of a delete action or simple negation to represent the changing state of the world. This type of active interference is referred to as *clobbering* in Chapman's lucid survey of planning with conjunctive goals [2].

3 Problem Description

The check for interference is performed by analyzing the static rules and integrity constraints of the knowledge base. This implies that the dynamic facts (i.e. specific slot values) contained in the knowledge-base at any given time are ignored. To detect interference between antecedents of a rule, we must examine the assumptions (missing dynamic facts) under which a rule will fire. These assumptions are not constrained by the current set of dynamic facts present in the knowledge base at a given point in time. Instead, any set of assumptions which is allowable via the semantic integrity constraints can be used. If it can be shown that inconsistent assumptions are required to satisfy a rule (i.e. an interference exists), than the rule will be unsatisfiable in a sound inference system. In the following sections, we present necessary terminology and definitions.

3.1 Proof-Residues and Ramifications

In [5] a resolution residue inference technique is presented. The byproduct of **resolution residue** on a goal is a set of missing assumptions which, if added to the knowledge base, would entail the goal. We refer to this set of missing assumptions as a **proof residue** of the goal. To determine a proof residue for a goal, we must generate a skeletal proof-tree of the goal using backward chaining. The proof-tree is skeletal, *since it represents a proof-strategy* rather than an actual proof that grounds out on stored facts. The leaves of a proof-tree constitute a proof residue. Since there may be multiple skeletal proof-trees of a goal, there can be several proof-residues for a goal. While constructing a proof-tree, the backward chaining is bounded by a specified proof-depth to ensure termination and to provide an upper-bound on the effort to expend.

It is important that a skeletal proof-tree be consistent within itself. After all, theoretically from inconsistent premises one could prove anything. We must use all available semantic typing information to ensure that none of following occur:

1) Assume a positive and negative instance of a literal. For example assuming $a \wedge \neg a$ is invalid.
2) Assume illegal inequalities. For example assuming $(?X > 10) \wedge (?X <?Y) \wedge (?Y < 5)$ is invalid.
3) Assume a set of conditions which violates a specified integrity constraint. For example, in DEVA the following integrity constraint can be specified to indicate that no port in a circuit shall possess both the value 0 and 1:
   $val(?Gate, ?Port, 0) \wedge val(?Gate, ?Port, 1) \rightarrow incompatible$
4) When dealing with systems that support frame-inheritance, assuming an object to be a member of disjoint classes. For example assuming instance-of(?X,human) and instance-of(?X,alien) is invalid.

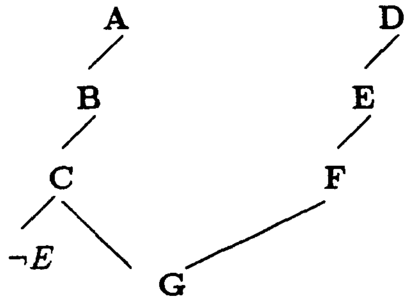
All literals derivable from a proof residue are **ramifications** of that proof residue. The original goal is a **necessary ramification** of the proof residue, as are all the intermediate literals in the proof-tree used in deriving the goal from the proof residue. There can also exist **extraneous ramifications** which are additional literals derivable via the proof residue, but which are unnecessary to prove the goal.

Definition 1 *Given two subgoals $\alpha \wedge \beta$ with proof-residues R^α and R^β respectively generated from consistent proof-trees, the proof-residues of the subgoals* **interfere** *if R^α and its ramifications are inconsistent with R^β and its ramifications.*

Consider the following rules and the subsequent skeletal proof-trees:

$C \rightarrow \neg E \qquad A \rightarrow B \quad B \rightarrow C$

$D \rightarrow E \qquad E \rightarrow F \quad C \wedge F \rightarrow G$



The proof-residue of the goal C is A. The ramifications of A are $\{\neg E, B, C\}$, where $\neg E$ is an extraneous ramification. The proof-residue of the goal F is D. The ramifications of D are $\{E, F\}$. Therefore an interference exists in the conjunctive goal $C \wedge F$, since the extraneous ramification $\neg E$ produced from proving C conflicts with a necessary ramification of the proof-residue of F. This example will also be referred to in the next section.

3.2 Weak and Strong Interference

For determining interference between antecedents, two types of interference checks are considered. If two antecedents of a given rule exist such that all consistent proof-trees of one of the antecedents interferes with any consistent proof-tree of the other, then we will have demonstrated an instance of **strong interference**. When strong interference is detected, the rule will never be satisfiable.

However, there may be times when finding interference between any two residues of two literals is of interest. This would be referred to as a check for **weak interference**. Assume that two subgoals each possess proof-residues generated from mutually consistent skeletal proof trees. By mutually consistent, we mean that the proof-residue and necessary ramifications of one sub-goal jibe with the proof-residue and necessary ramifications of the other sub-goal. If an extraneous ramification of one subgoal's proof-residue is inconsistent with some ramification of the other subgoal's proof-residue, we have a potential rule-inconsistency problem. This is the situation evident in the example of the previous section. The subgoals $C \wedge F$ possess mutually consistent proof-trees. However if a situation is created in the fact-base to exercise these proof-trees, the inconsistency $E \wedge \neg E$ will be created.

Sometimes strong interference is a desirable property of two literals. DEVA allows the user to specify ad-hoc constraints on the behavior. For example, the designer would be allowed to specify the rule $\alpha \wedge \beta \rightarrow$ *incompatible* to indicate that the conditions $\alpha \wedge \beta$ should never be true. If it could be proved that a case of strong interference existed for constraints of this form, then we will have validated the knowledge base against these constraints.

4 Monotonic Interference Detection

4.1 Monotonic Backward Chaining Detection

For illustrative purposes, it is useful to examine the monotonic case of backward-chaining interference BCI) detection first. The method for detecting interference in the monotonic case is relatively straight-forward but it is an integral part of generating a consistent residue for any nonmonotonic method. Related techniques [1,12] have been forwarded that make use of semantic information to optimize user-level queries in deductive databases by, among other things, determining when such queries are unsatisfiable. Backward-chaining interference detection in effect treats the LHS of each rule in the knowledge base as such a query. The emphasis here is not so much on user-level query optimization, but instead on highlighting possible anomalies for each respective rule.

The method begins by examining the LHS of each rule in the KB as a conjunctive goal. This is referred to as the original LHS, to differentiate it from the LHS's of other rules involved in the process of proving the original LHS.

The residue of each antecedent is generated by backward inference from the antecedent to the base set of assumptions necessary for it to be true. The construction of the proof residue is interleaved with the check for interference. Whenever the entire LHS of a rule is satisfied during this inference, the proof tree is updated with the current RHS goal occurring in that rule. Then this goal's residue and necessary ramifications are examined to see if they contradict the residues and necessary ramifications of any of the other original LHS goals (the types of possible conditions are outlined in the section dealing with residues and ramifications). If such a contradiction is found, it is reported to the user as a case of weak interference. No further exploration of that particular proof residue is then necessary. Otherwise, the inference proceeds to the maximum allowed depth, trying to satisfy the rest of the goals. Detecting strong interference under this scheme is a matter of recording the presence or absence of interference in each alternative proof residue attempted. If every such proof residue attempted has weak interference, then the developer is warned of the presence of strong interference for the respective rule.

It is sufficient to consider only the original LHS when looking for contradictions in each proof residue. We can ignore possible interference in the other LHS goals involved in the proof residue. The interference check iterates through all of the rules in the knowledge base so these other LHS goals will eventually be examined as the original LHS and any interference will be detected at that time.

The backward-chaining method is capable of detecting the strong interference that is present in the following monotonic circuit example (See Figure 3 and accompanying rule base).

Suppose we wish to test to see if gate's output port can be tested for a stuck at zero error. This involves generating a test pattern across the circuit inputs that will manifest a value of 1 at the circuit's output. This is essentially a residue of $val(?input, out, ?val)$ facts. Whenever we use the test pattern as input on a real version of the circuit and a one appears at the output, we know the output port is not stuck at zero.

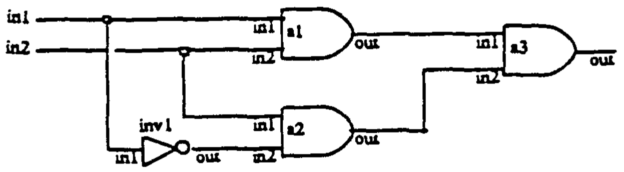


Figure 3: Anomalous Circuit.

$FACTS:$

$type(i1, input)$	$type(i2, input)$	$type(a1, and)$	$type(a2, and)$
$type(a3, and)$	$type(inv1, inv)$	$conn(i1, out, a1, in1)$	$conn(i2, out, a1, in2)$
$conn(i2, out, a2, in1)$	$conn(i1, out, inv1, in1)$	$conn(inv1, out, a2, in2)$	$conn(a1, out, a3, in1)$
$conn(a2, out, a3, in2)$			

$RULES:$
$val(?gate, ?port, 1) \rightarrow can\_test\_stuck(?gate, ?port, 0)$
$val(?gate, ?port, 0) \wedge val(?gate, ?port, 1) \rightarrow incompatible$
$type(?gate, and) \wedge val(?gate, in1, 0) \wedge val(?gate, in2, 0) \rightarrow val(?gate, out, 0)$
$type(?gate, and) \wedge val(?gate, in1, 1) \wedge val(?gate, in2, 0) \rightarrow val(?gate, out, 0)$
$type(?gate, and) \wedge val(?gate, in1, 0) \wedge val(?gate, in2, 1) \rightarrow val(?gate, out, 0)$
$type(?gate, and) \wedge val(?gate, in1, 1) \wedge val(?gate, in2, 1) \rightarrow val(?gate, out, 1)$
$type(?gate, inv) \wedge val(?gate, in1, 0) \rightarrow val(?gate, out, 1)$
$type(?gate, inv) \wedge val(?gate, in1, 1) \rightarrow val(?gate, out, 0)$
$conn(?gate1, ?port1, ?gate2, ?port2) \wedge val(?gate1, ?port1, ?val) \rightarrow val(?gate2, ?port2, ?val)$

The BCI method begins by examining the first rule, which indicates that a gate's port can be tested for a stuck at zero condition. It will find that the LHS goal val(a3,out,1) is unsatisfiable because LHS literals $val(a3, in1, 1) \wedge val(a3, in2, 1)$ interfere with each other. Backward-chaining will first determine that the residue for $val(a3, in1, 1)$ is $\{val(i1, out, 1), val(i1, out, 1)\}$. Then the goal $val(a3, in2, 1)$ will be examined. Its residue is $\{val(i1, out, 0), val(i1, out, 1)\}$. This will be reported as a case of weak inteference, and since there are no other alternative, consistent residues, it will be reported as a case of strong interference as well. In general, it is impossible to find a test pattern for this circuit that will manifest a 1 as the output of $a3$ because of the inverter $inv1$. The output of $a3$ will always be zero.

4.2 Monotonic Forward Chaining Detection

The monotonic backward-chaining interference detection method of the preceding section filters out the internally inconsistent skeletal proof-trees of a conjunctive goal, where the conjunctive goal is simply the antecedents of a specific rule. The user is informed of the presence of interfering proof-trees and can browse them at leisure. If there does not exist an internally consistent skeletal proof-tree for the conjunctive goal, then a case of strong interference exists for the conjunctive goal. If an internally consistent skeletal proof-tree exists for the conjunctive goal, then we know that the proof-residue of the conjunctive goal and its necessary ramifications are consistent. We still need to verify the absence of a potential inconsistency caused by the extraneous ramifications. To compute the extraneous ramifications, we need to simulate forward-inference from the original proof-residue. This requires symbolic execution, since the variables may only be partially constrained (remember, a skeletal proof-tree is a proof strategy rather than an actual grounded proof). To guarantee termination, an upperbound can be specified to indicate the number of levels of forward inference to perform. This also allows the user to choose a degree of completeness for the verification.

5 Nonmonotonic Interference Detection

Ordinary logic is monotonic in nature. Any deduction from a base set of facts and conditionals will only result in adding more facts to the base that were already implicit. Nonmonotonic systems allow revision of an existing state S, and it is this revision that can lead to contradictions [4]. The closed world assumption (CWA) completes a given state S by assuming that if a literal α cannot be proven, then $\neg\alpha$ is assumed true. Since the knowledge base may be revised in some subsequent state, α may be provable later. This can be another source of interference, since the proof-tree may expect a literal to be present and absent during the same knowledge base state. Consider a pair of antecedents $\neg\alpha \wedge \beta$, and a set of rules where α is always proved as a consequence of proving β. Since satisfying β will always result in α, the pair $\neg\alpha \wedge \beta$ should never be true in the same KB state S. Another form of revision in nonmonotonic systems are explicit delete actions, because instead of bringing a new fact into the KB at a later time, an existing fact is removed. Some of the propositions in S may rely on α, so they may have to be revised as well.

In the presence of nonmonotonicity and the absence of truth maintenance, inconsistent knowledge base states can be generated, and as a result rules with interference problems may be satisfiable. A truth maintenance system (TMS) which maintains information on whether or not literal α is currently true in a state S can maintain the correct declarative context for conjunctive goals because the conjunctive notion of KB state is achieved by reconsidering the assumptions upon which a given proposition relies [3]. However, the computational overhead of using an underlying TMS may be prohibitive [9,5]. Many expert systems are implemented without an underlying TMS, or allow rules outside of the control of the TMS. For these systems, DEVA's interference detection will warn the system designer when rule satisfiability can occur only via procedural and unsound inference (which may be allowed via the production rule semantics). Additionally, many implementations for truth maintenance do not provide any assistance to the developer when the source of interference must be located [10]. So for these systems, DEVA will act as a "diagnosis" tool to explain why a certain rule "did not work".

In nonmonotonic knowledge bases, it can be shown that forward-chaining and backward-chaining do not necessarily give the same results. Therefore to accurately predict interference, the evaluation strategy employed by the expert system must be modeled. To see this, consider the following example, where the facts *Shoot* and *Loaded* are true:

Forward-Chaining Production Rules	Backward-Chaining Prolog Rules
r1: $Loaded \wedge Shoot \rightarrow delete(Loaded) \wedge Dead$	r1p: $dead : -loaded, shoot, retract(loaded).$
r2: $Dead \wedge Loaded \rightarrow Goal$	r2p: $goal : -dead, loaded.$
r3: $Loaded \wedge Dead \rightarrow Goal$	r3p: $goal : -loaded, dead.$

Here the *Loaded* literal means a single-shot gun is loaded, the *Shoot* literal indicates the gun is fired, and *Dead* means the target is dead. $r1$ is the rule that creates the new knowledge base state resulting from the shooting. The rules $r2$ and $r3$ are order variants of a desired goal: we somehow want the target dead and the gun to remain loaded. The literals *Dead* and *Loaded* can both be satisfied separately in different knowledge base states. Yet considered together within the same knowledge base state, they are unsatisfiable. Under forward chaining inference where right-hand-side actions occur in parallel, neither $r2$ nor $r3$ will fire.

With backward chaining and the left-to-right (LR) evaluation of the antecedents, prolog rule $r3p$ will satisfy the goal. Prolog rule $r2p$ will not satisfy the goal, because satisfaction of the subgoal *dead* interferes with satisfaction of the subgoal *loaded.* As a consequence of proving the subgoal *dead*, the fact *loaded* is retracted from the database.

5.1 Backward Chaining Detection

Backward-chaining interference detection can be augmented in the presence of nonmonotonicity with only a small overhead by incorporating a state representation that captures the effects of local extraneous ramifications. The rule $A \wedge B \rightarrow D, E, delete(F)$ appearing in a knowledge base can be equivalently expressed as three separate rules:

$A \wedge B \rightarrow D \qquad A \wedge B \rightarrow E \qquad A \wedge B \rightarrow delete(F)$

Thus when proving D, its local, extraneous ramifications are E and the deletion of F. If a separate rule $A \rightarrow G$ appears in the knowledge base, then G is considered a *nonlocal* extraneous ramification of D.

This method relies upon a fixed evaluation strategy and concept of state. For most knowledge base systems that offer backward-chaining, the state S is cumulative according to goal ordering. Whenever a goal is proven, its local ramifications are used to update the state. For backward chaining in expert systems, conjunction between antecedents is usually implemented by sequentially examining the antecedents. Once an antecedent is satisfied, the next is examined in the same manner. Even though satisfying the next antecedent may cause the previously examined antecedent to become unsatisfiable because of nonmonotonic properties, the previously satisfied literals are not re-examined to see if they are still true in the final state S. This is fine for a procedural interpretation, but when the antecedents of a rule are stated conjunctively in a declarative context, they must all be satisfiable in the same state of reasoning.

Essentially interference detection is achieved by cumulatively updating the state and re-examining previously satisfied literals to see if they still hold true. This form of detection indicates when rules are unsatisfiable, and highlights the case where the LHS of a rule is procedurally satisfiable, but its declarative context is not upheld.

The antecedents of each rule in the knowledge base are examined in the same manner as in the monotonic case of backward-chaining interference detection. The backward inference proceeds in an attempt to satisfy each of the goals in the LHS in left-to-right order. Whenever the antecedents of a rule necessary for the proof are satisfied during this inference, the state representation, in the form of a proof residue, is updated with the goal proven and any local ramifications (or side-effects, in the case of Prolog). This is a limited form of forward chaining local to the rules necessary for the proof. At this point, the state representation is examined to see if it contradicts any of the original LHS goals. If such a contradiction is found, it is reported to the user as a case of weak interference.

The advantages gained from interleaving the construction of a proof residue with state update and interference check is that no separate pass over the proof residue is required to detect contradictions. Other advantages are those inherited from normal backward inference: only those rules which participate in the proof of the original LHS are examined, increasing efficiency. Since all these rules directly take part in proofs, coherent explanation traces of how weak interference occurs can be easily collected for presentation to the developer.

A disadvantage of this augmented backward-chaining detection is that it only takes into account local extraneous ramifications appearing in the RHS of rules actually involved in the proof-residue. This is entirely appropriate for backward-chaining inference systems, but there are other ramifications possible under forward inference. These extraneous ramifications can lead to contradictions as well, and under systems which have coherent forward inference, this check *may not detect all cases of interference*. However, determining all the ramifications of a given residue is an intractable problem in the general case. Some upper bound on the effort to expend must be set. The tradeoff is completeness versus computing time. In this case the upper bound is limited to local, extraneous ramifications so that a reasonably quick check for interference is available for the developer, even if he is relying on a foward inference system. Under the forward-chaining method that follows, the consequences of all ramifications can be evaluated to a given level of forward inference, providing the developer with a more complete check.

5.2 Forward Chaining Detection

To analyze nonmonotonic rule-bases for interference, we need to model the operational semantics of rules possessing side-effects. The intuitive reading of the rule $LHS \rightarrow RHS$ is, 'Whenever a knowledge base state *S1* can be achieved where LHS is true, perform the actions in RHS to create a new knowledge base state *S2*. All conditions true in *S1*, not explicitly changed via RHS, are assumed to persist in *S2*.' This is the STRIPS assumption, which is based on the situation calculus. During forward chaining from some knowledge base state, several rules may be satisfiable. We assume a *global* update policy where all satisfiable rules fire in parallel to create a new knowledge base state. This strategy is actually required to maintain a declarative context for forward chaining [16]. Some expert systems assume an *immediate* update policy – as soon as a rule is deemed satisfiable, its RHS actions are performed. This is efficient but blurs the notion of KB state, since the rule's RHS actions can pre-empt the firing of others which were satisfiable before the RHS actions were performed.

To motivate our discussion, consider the following propositional rules from a contrived air traffic control KB:
$R101 : Safe \wedge Cleared \rightarrow Land, delete(Holding)$
$R102 : Holding \wedge RunwayAllocated \rightarrow Cleared$
$R236 : Fire \wedge Holding \rightarrow Cleared, delete(Safe)$

This knowledge base demonstrates how weak interference can lead to unexpected failure of proofs. Let us suppose that Frank and Earnest are KB developers. Initially Frank adds $R101$ and $R102$ to the KB. These rules express the normal, safe preconditions for landing. Later Earnest decides there may be some overriding conditions, such as a plane on fire, which would allow clearance, and so he adds $R236$ elsewhere in the KB files. This introduces an example of weak interference, which will present a severe problem in case a plane catches on fire during its holding pattern (since no runway will be allocated unless the conditions are safe). Even though the developers may think they have covered this situation, they have not. By showing the case of weak interference, Frank and Earnest will be prompted to reexamine the rules.

To reiterate, the first step in interference detection is the generation of a consistent proof-tree for each subgoal. In the monotonic case, there is only one KB state and all inference is done with respect to that state. In the nonmonotonic case a literal can be derived from rule-interactions over several KB states, with rule satisfaction occurring in one state and the rule's actions resulting in another. Therefore, in the nonmonotonic case, a proof-tree reflects several KB states with each node representing a different KB state derived from the union of its parents' KB states. To compute the extraneous ramifications, we now need to simulate forward-chaining inference over the KB-states represented by the proof-tree.

Two proof-trees exist to satisfy rule $R101$'s preconditions. See Figure 4. The first proof tree represents an interaction between $R101$ and $R102$, while the second proof tree represents a weak interference resulting from an interaction between rules $R101$ and $R236$. In Figure 4 the proof-residue and its necessary ramifications are displayed in boldface. The non-boldfaced literals at each node (KB-state) are either literals persisting from a prior state or extraneous ramifications computed via forward-chaining from the prior state.

So for example in the second proof-tree, a KB-state containing $Fire$ and another containing $Holding$ were merged and forward-chaining was performed on the union. Only rule $R236$ applies, and consequently a new KB-state is created reflecting the actions of this rule (namely to clear the plane and declare the situation unsafe). If other rules had applied, all rules would have been fired in parallel to create the new state.

In the second proof-tree, an interference exists between the proof for $Cleared$ and the proof for $Safe$. This is reflected in the final KB-state which requires both $Safe \wedge \neg Safe$. These proof-trees also demonstrate how truth-values can change over several KB-states. Both proof-trees require $Holding$ to be true initially, but false after $Landing$ has been derived. As suggested by this example, interference can be detected by simply checking the individual KB-states for inconsistencies. Incidentally as an implementation note, the work in [6] directly

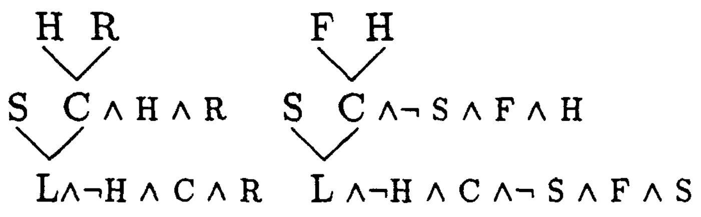


Figure 4: Air Traffic Proof Trees

supports our need to model persisting and changing values over a partial ordering of multiple states.

5.2.1 Modelling Negation by Failure

We have not addressed the problem of handling negation by failure in the antecedents of rules. Some of the assumptions within a proof-residue may be statements about the unprovability of a goal. Before deeming a skeletal proof-tree internally consistent, we need to rectify the unprovability assumptions with the other assumptions in the proof-residue. For example, the proof-residue $\{B, C, unprovable(E)\}$ of some conjunctive goal seems internally consistent. If the rule $B \wedge C \rightarrow E$ exists, then this proof-residue is not internally consistent.

6 Summary

The presence of interference between antecedents can lead to anomalies in a knowledge base application, such as rule unsatisfiability, different behavior for forward and backward inference, unexpected failure of proofs, and sources of rule-inconsistency.

When dealing with monotonic rule-bases, we provide two search techniques for uncovering interference. An efficient goal-directed search technique can be employed to uncover an incomplete, yet revealing, subset of all possible interferences. By integrating an expensive forward-chaining search technique with the goal-directed approach, the remaining interferences can be uncovered.

When dealing with rule-bases containing arbitrary side-effects (i.e. deletes) in the consequents of rules, correct and meaningful detection of interference is dependent upon the evaluation strategy used. Therefore we provide two methods of detection; a backward-chaining left-to-right method for systems similiar to Prolog, and a forward-chaining global update method. The mixed strategy of inference, using both forward and backward chaining, is not addressed because the concept of KB state becomes blurred under such systems and intuition suggests that many of the significant cases of interference can be detected by one of the two methods outlined in this paper.

Acknowledgements Many thanks to Jay Weber, Bill Mark, Rich Pelavin, and Jacky Combs for their insightful comments.

References

[1] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2), June 1990.

[2] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, July 1987.

[3] J. de Kleer. An assumption-based tms. In Matthew L. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, Morgan Kaufmann, 1987.

[4] Jon Doyle. A truth maintenance system. In Matthew L. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, Morgan Kaufmann, 1987.

[5] Joseph Jeffrey Finger. *Exploiting Constraints in Design Synthesis.* PhD thesis, Stanford University, 1987.

[6] Tim Finin and Jim McGuire. Inheritance in logic programming knowledge bases. In M. Lenzerini, D. Nardi, and M. Simi, editors, *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, John Wiley and Sons, 1990. From Workshop on Inheritance Hierarchies, Viareggio, Italy, 2/89.

[7] A. Ginsberg. Kb reduction: a new approach to checking kb's for inconsistency and redundancy. In *AAAI*, 1988.

[8] Allen Ginsberg and Keith Williamson. *Checking Quasi-First-Order-Logic Rule Based Systems for Inconsistency and Redundancy.* Technical Report 11354-891229-02TM, AT&T Bell Labs technical memo, December 1989.

[9] Henry A. Kautz and Bart Selman. Hard problems for simple default logics. In *First Intl. Conf. Principles of Knowledge Representation and Reasoning*, ACM, Morgan Kaufmann, 1989.

[10] J. P. Martins and S. C. Shapiro. A model for belief revision. *Artificial Intelligence*, 35(1):25–79, May 1988.

[11] Jim McGuire. Uncovering redundancy and rule-inconsistency in knowledge bases via deduction. In *Fifth Annual Conference on Computer Assurance: Systems Integrity and Software Safety (COMPASS-90)*, IEEE, June 1990. Also available as Lockheed techreport AIC-89-106.

[12] J. R. McSkimin. *Techniques for employing semantic information in question-answering systems.* PhD thesis, Dept. of Computer Science, Univ. of Maryland, 1976.

[13] S. Raatz and G. Drastal. Patterns of interaction in rule-based expert system programming. *Computational Intelligence*, 3:107–116, 1987.

[14] R. Stachowitz, C. Chang, T. Stock, and J. Combs. Building validation tools for kb systems. In *1st Annual Workshop on Space Operations Automation and Robotics, Houston*, 1987.

[15] R. Stachowitz and J. Combs. Validation of expert systems. In *20th Hawaii Intl. Conf. on System Sciences*, 1987.

[16] R. Treitel and M. Genesereth. Choosing directions for rules. *Journal of Automated Reasoning*, 3(4):395–432, 1987.

[17] R. J. Waldinger. Achieving several goals simultaneously. In B. L. Weber and N. J. Nilsson, editors, *Readings in Artificial Intelligence*, Morgan Kaufmann, 1981.

SIMULATION IN SUPPORT OF SPECIFICATION VALIDATION

Kevin M. Benner
USC / Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
(213) 822-1511

1 Abstract

Current techniques for dynamic validation of formal specifications have failed to scale to real world applications. This paper will describe why this has been the case and why a combination of symbolic evaluation and simulation will scale. Additionally, this paper will describe progress at implementing SIMSYS, a simulation tool, one part of a dynamic validation suite. Specific problems addressed within SIMSYS include incompleteness, largeness, and nondeterminism.

Keywords: dynamic validation, simulation, symbolic evaluation, nondeterminism, scenarios, and approximate models.

2 Introduction

During the development of specifications, a variety of validation tools and analysis techniques must be brought to bear. These tools/techniques focus on various aspects of the specification. This paper focuses on those aspects which are concerned with the execution semantics of the specification. This class of techniques is often referred to as dynamic analysis and may include: symbolic execution, executable specifications, and simulation. Up until now there have been fundamental problems which have prevented their large scale use. Symbolic evaluation has been restricted to small specifications because state descriptors get too large and unwieldy in larger specifications [9]. Executable specification languages (e.g., [14], [2], state transition diagrams, and petri nets) have been limited to restrictive, less expressive languages. And simulation has resorted to implementation like languages (e.g. SLAM, lisp and C) to describe simulation behavior [12].

The purpose of the work described here in has been to build a tool for dynamic validation which will scale to large, complex specifications written in rich specification languages. Such a tool must allow user guidance during the validation process without requiring excessive amounts of guidance. Such a tool must provide a reasonable level of assurances about the dynamic behavior without requiring testing of every possible behavior. And finally, such a tool must be tightly coupled with the rest of the specification development and validation process.

This paper will describe why individual techniques have not scaled and why a combination of symbolic evaluation and simulation will scale. Additionally, this paper will describe progress at implementing SIMSYS, a simulation tool, one part of dynamic validation suite. Specific problems addressed within SIMSYS include incompleteness, largeness, and nondeterminism.

The emphasis in this work has been to provide a general purpose simulation facility, currently called SIMSYS, which works within the ARIES environment [8]. Given ARIES ability to support multiple requirement and specification methodologies and languages, it is believed that the techniques described are fairly general purpose for simulating high level specification languages. In summary, behaviors which can not be handled directly by the compiler are either guided interactively by the user or by user-provided scripts called scenarios. The simulation system provides a highly interactive environment which supports rapid execution, modification, and re-execution. Mechanisms are provided to focus the level of detail simulated within the simulation. There are additional mechanisms to show selected presentations of the executing specification to the user.

3 Vision

Symbolic evaluation has been effective at proving for an entire class of inputs whether or not the specified system will enter an illegal state – as characterized by invariants describing user requirements. This has been done by deducing successive states in the behavior space. The problem with large specifications has been that the state descriptors get too large and unwieldy to manipulate and understand easily.

Simulation has been effective at guaranteeing satisfiability of the specification with respect to specific test cases. Simulation results, because of their concrete nature, are directly understandable by the user and hence easily validated with respect to user intent. This, though, does not make any generalized claims about the validity of the specification.

The marriage of these two techniques provide interesting possibilities. In particular, symbolic evaluation may use simulation. That is, simulation results can be used to simplify symbolic evaluation state descriptors. This would result in state descriptors which are now possibly of reasonable size and amenable to additional reasoning. Additionally, simulation may use symbolic evaluation. That is, during simulation the user may ask Is-possible questions which the symbolic evaluator may try to answer with respect to the current state of the simulation.

A typical session with such a combined tool would consist of the developer interacting with the simulator in order to explore the space of possible behaviors. Based on the developer's insight – gained from previous knowledge of the domain and interaction with the tool – the developer will evolve his/her intuition about where in the behavior space interesting behaviors may occur. The desire is to allow the developer to get to this point in the behavior space and then explore it at various levels of detail using both case-based and generalized analysis techniques. Case-based analysis techniques refer to the use of user provided scenarios and concrete simulation to validate via user observation that desired behaviors are occurring. Generalized analysis techniques refer to the use of symbolic evaluation to formally prove that inconsistent states are not entered. Such inconsistencies would be violations of stated formal requirements.

Progress to date includes a symbolic evaluator [3] [4] which works well on small specifications and SIMSYS. The remainder of this paper describes the problems that had to be dealt with to build SIMSYS and the solutions to those problems.

4 The Problem

The very nature of high level specifications make them difficult to simulate. By desire, they typically describe what behaviors are described without stating how they are to be achieved. Features of a specification that are problematic include: scale, incompleteness, and high-level language constructs.

A simulator must have mechanisms to allow it to scale to handle real world specifications which are large and complex. A simulation of such specifications may be prohibitively slow if executed in full detail. Conversely, a simulation may result in incorrect validation if not executed in enough detail (i.e., due to missing low level interactions). Furthermore, the user may be overwhelmed with too much information if it is not adequately controlled.

Because a specification is under development, it is inherently incomplete. Incomplete means terms are not defined or are only partially defined. Additionally, a specification may be incomplete in the sense that everything has not been defined to the desired level of detail (i.e., a highly abstract specification). In spite of this, the specification must still be simulatable at any point during the development.

High-level constructs provide to the specifier (user) specification freedoms along various dimensions. These constructs include nondeterminism, historical reference, and high level data structures. The problems posed by some of these constructs have adequate solutions. For example, high level data structures may be compiled using techniques described in [5], [7], and [10]]. Other constructs, in particular nondeterminism, are still problematic.

4.1 Nondeterminism In Greater Detail

A specification can be said to describe a set of behaviors. More accurately these behaviors should be thought of as a tree with each node representing a state and each arc representing an action which will advance the system to some new state (i.e., different node). In order to focus on some aspect of a developing specification a designer may temporally ignore other aspects of it. The specifier instead relies on angelic nondeterminism to allow the "right" things to happen during execution (i.e. the specifier delays fully specifying control mechanisms, but assumes correct behavior will still occur)

Below is a simple example of angelic nondeterminism with respect to a person sitting down.

```
invariant 0-or-1-per-chair
  for-all (chair) ( count (occupied(chair, ?)) LE 1)

procedure sit-down[person]
  insert occupied(ANY CHAIR, person)
```

ANY CHAIR is a nondeterministic reference. The invariant says that only 0 or 1 people may occupy a given chair. Therefore, when the procedure SIT-DOWN is invoked the result should be that the person should OCCUPY any empty chair. Angelic nondeterminism allows us to assume that is what happened without providing a selection function for ANY CHAIR that would select only empty chairs.

In terms of the behavior space tree, nondeterminism may be visualized as a tree with too many arcs (i.e., the unnecessary or undesired arcs describe actions which should be illegal). With respect to the example, the action of sitting would result in a distinct arc for each chair, thus representing all the actions possible for sitting down. The selection of an arbitrary path could result in a person sitting in a chair which is already occupied. This would be illegal since it violates the invariant. The angelic type of nondeterminism ensures that if there exists a legal path through the tree, then it is this path to which the specification refers. If there are multiple legal paths, this would be normal nondeterminism.

Though the ultimate goal of the development process is to eventually remove all the unwanted arcs, the specifier may not be ready to do this now for a variety of reasons. Given that many of the arcs are illegal, the simulator still needs to be able to traverse the behavior space tree.

The issue then is how to make choices at each node to allow the desired behaviors to occur. Operationally, angelic nondeterminism could be attained via backtracking. Assuming that the current and goal states are each known, an execution environment could make arbitrary choices at each node. If it ends up in a state which cannot be advanced toward the goal state the system backtracks to a previous node and makes a new choice. Though elegant, this approach fails for two reasons. One, it does not scale to large specifications which could have a huge behavior space. And two, with regard to validation, a user will find this backtracking behavior very non-intuitive in terms of describing what is going on (i.e., sort of like watching a movie which is sporadically going forwards and backwards over different plot lines and then expecting the viewer to tell you what the movie was about).

4.2 An Example

The concepts described in this paper will be described with respect to the following example. Due to a lack of space, the amount of Gist will be kept to a minimum.

The application domain will be a traffic light which controls a single intersection of a north-south road and an east-west road. Each road contains a left turn lane and a straight lane in each direction. Each lane has its own traffic light.

The requirements of the traffic light are that cars be able to pass through the intersection in a safe and timely fashion. Safe is formalized as acceptable signal colors for each light at the intersection. Timely will not defined in this example.

An early design commitment is that the traffic light run on a timer as well as respond to the presence or absence of vehicles.

The initial specification of the traffic light controller is via demons which change the signal color of individual lights. Below is one such demon. The actual specification has analogous demons for traveling straight in each direction and turning left in each direction.

```
    demon change-traffic-flow-to-east
        [int:intersection, tl-e: traffic-light]
           when   traffic-light-direction(tl-e, 'east) and
          signal-color(int, tl-e, 'red) and
        elapsed-time(tl-e, 60) and
waiting-cars(int, tl-e)
       := change-traffic-flow(int, tl-e);
```

This is a plausible specification which may be directly validated by the user. Missing from this specification is control information on how signals change. Some of this information may be in demon preconditions, but this must be both validated (i.e., does it satisfy the requirements on safe traffic flow) and augmented (i.e., elaborated so that the traffic light operates in a deterministic fashion). The goal of dynamic validation is to aid this process.

5 Basic Solution

5.1 Incompleteness

During the specification development process it is not unusual for some terms to be undefined. Despite this fact, one must still be able to simulate the specification. Zave in PAISLey [14] handles undefined terms by providing interactive facilities which query the user to provide the missing resultant value. Alternatively, since in PAISLey all terms are functions, if at least a type signature is available, PAISLey can be configured to select an arbitrary value from the range of the partially defined term. The simulation community [11] [15] provides a richer set of alternatives. One technique consists of defining an ill-defined term with respect to an existing term which approximates the desired semantics. The idea here is not to fully define the term but to provide an approximation which is good enough for use within the context of a specific simulation.

SIMSYS incorporates these ideas by providing to the user a full range of approximation models which may be used within the context of the simulation to define ill-defined terms. The desire is that the user will select the model that adequately approximates the desired behavior necessary to run the simulation. This facility is necessary because non-trivial specifications have multiple components that interact. Dynamic validation is often delayed because nothing can be done until all interacting components are described. Approximations allow the user to quickly describe some components and then focus on other components in greater detail. Dynamic validation is now available at a much earlier stage of the development.

As an example consider the relation waiting-cars used in the demon above. Rather than defining it now as a temporal relation between a traffic signal and the location of some cars, the user may approximate it via a stored relation which is directly asserted by the user. This then allows the user to write a simple generator which directly assert that cars are waiting. This is a good enough approximation to allow the user to focus on the control flow for changing traffic signal colors. Conversely, when the user is ready to define waiting-cars he/she can replace the approximation with the actual definition.

There is no attempt to directly incorporate these approximation models into the specification. This is because the decision on the suitability of a model is driven by the very narrow needs of the current simulation, rather than the more comprehensive needs of a system specification. This is demonstrated in the above example, where the stored relation approximation does not embody any real notion of what waiting-car means.

How to find an acceptable approximation in a large library has not been addressed yet. The current approximation library is relatively small consisting of primitive concepts like the one in the above example. Eventually this library should have higher level concepts, analogous to cliches [13] (e.g., schedule, generate, filter, communicate, monitor, etc.).

5.2 Scale

Dealing with large specifications is a matter of scoping and focusing. This is handled in three ways: selecting which components to include, deciding what level of detail to simulate at, and focusing on specific sequences of behavior.

Responsibility for selecting which components to load is split between the user and SIMSYS. The user selects those components he/she is specifically interested in validating. SIMSYS identifies interfaces between these components and the external environment and ensures that external models exist to drive them. Together this is referred to as the execution model.

At this point the execution model is traversed by SIMSYS. Components already defined in the execution model are used as is. When a component is not defined in the execution model, the user is queried to load the specification definition or some approximation of it. Approximations may be like those described in the previous section or they may be generalizations or specializations of the component. Depending on what aspect of the specification the user is validating, he/she decides the appropriate level of abstraction to use for each component.

As well as deciding on these gross abstraction layers, the user can also focus on specific relations (to the exclusion of others). An example of this is to focus the traffic light specification so that it will be simulated with respect to only the presence of cars or conversely with respect to only the elapse time since the traffic signal last changed. Being able to separate behaviors along these lines allow the user to first validate more primitive behaviors before dealing with the more complex behaviors.

The last issue with respect to scale is what should the user see during the simulation. Even though the simulator has been focused to some set of activities, there may still be lots of things going on at one time. The user must be able to select what things to view and how to view them. In SIMSYS this is done via automation rules which can be set to watch for the occurrence of specific conditions. Upon seeing the condition they notify the user or the system. This may take the form of textual output or updates to graphical presentations (e.g., state transition diagram or domain specific presentations)

5.3 Nondeterminism

Even with scoping and focusing mechanisms, nondeterminism will still be present and must be handled. As stated earlier, Angelic nondeterminism can be implemented via backtracking, but is less than satisfactory because of the potentially large behavior space trees that result from a complex specification and the non-intuitive nature of the resulting execution. To reduce (hopefully eliminate) the need for backtracking, the simulator attempts to make the "right" choice.

In conventional simulation systems, the user makes implementation commitments which remove nondeterminism enabling the specification to be simulated. This is a problem because the implementation commitments are either directly embodied into the specification [] or are realized via implementation code [15]. Both of these approaches violate the spirit of KBSA development by forcing the developer out of the specification space into nitty gritty implementation details needed to make the specification executable. One runs the risk of confusing commitments in the specification which are motivated by system requirement with those motivated by simulation requirements. Clearly only the former should be present. To preclude this confusion, one needs to be able to remove nondeterminism at the specification level without influencing the specification under development. SIMSYS provides three approaches to do just this. One approach is to interactively ask the user during simulation. Another is to use probabilistic models to resolve the choice. And a third approach is for the user to describe these choices beforehand.

Interactive choice provides the greatest flexibility, but if relied on exclusively would typically overwhelm the user with questions. Probabilistic models are also common within the the simulation community and in some cases are an ideal choice. When applicable the first two techniques have been effective.

Descriptions of a priori choices have been relatively simple, consisting basically of precanned answers to interactive questions. A more declarative approach, known as scenarios, is used by Fickas [6] for partially describing cases within his critiquing environment. In [1], Johnson and I describe why scenarios are useful during the specification development process and provide a notation for describing them. For the problem at hand scenarios are useful because they are easy to capture and validate and can be used to prune arcs within the behavior space tree. With respect to this use, scenarios can be defined as a partial ordering of events which describe some sequence of events that the user is interested in.

Consider the demons described earlier which change traffic flow. The current state is signal-color('Intersection-1, 'Traffic-Light-North, 'Green) and signal-color('Intersection-1, 'Traffic-Light-South, 'Green). All other traffic signals are red. Ignore the relation elapse-time. Assume cars are waiting at all red traffic signals.

Given this focus and initial scenario, the preconditions of each demon associated with traffic signals which is red are satisfied. Each demon is an arc within the behavior space to a new state. Paths along some of these arcs will lead to requirements violations (i.e., safety violations).

Rather than deal with the control of all demons at once, the user may provide scenarios which narrow the behavior space. One such scenario would say that all north, north-left, south, and south-left change-traffic-flow demons should, if enabled, execute once before executing the east, east-left, west, and west-left change-traffic-flow demons. This scenario segments the traffic flow demons into two groups. Control within these two groups is still nondeterministic, but the overall amount of nondeterminism at each state has been cut in half.

At this point the amount of nondeterminism at any node has been reduced enough so that it may be handled interactively by the user. Alternatively the user may provide additional scenarios which further sequence the change-traffic-flow demons within a given group. Now the goal for the user may be to discover the optimal sequencing of traffic signals which do not violate requirements and handle the anticipated volume of traffic. This can all be done via conventional simulation analysis techniques within SYMSIS.

6 Conclusion

The basic building blocks for dynamic simulation have been built. Individually, each symbolic evaluation and simulation have their particular strengths and weakness. Current capabilities allow both of these validation techniques to be applied to the same specification. Additional work remains to realize the Vision described earlier of integrating these systems to allow the results of one tool to be used by the other.

References

[1] K.M. Benner and W.L.Johnson. The use of scenarios for the development and validation of specifications. In *AIAA Computers in Aerospace VII Conference, Monterey, CA*, 1989.

[2] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[3] D. Cohen. Symbolic execution of the gist specification language. In *Proceedings, IJCAI-83*, 1983.

[4] D. Cohen. A forward inference engine to aid in understanding specifications. In *Proceedings, AAAI-84*, 1984.

[5] D. Cohen. Automatic compilation of logical specifications into efficient programs. In *Proceedings, AAAI-86, Philadelphia, PA, Aug. 1986*, 1986.

[6] S. Fickas and P. Nagarajan. Automating the transformational development of software. *IEEE Software*, pages 37–47, November 1988.

[7] et al Goldberg, A. Progress on the kbsa performance estimation assistant. In *3rd Annual KBSA Conference, Utica, NY, Aug. 1988*, 1988.

[8] W.L. Johnson and D.R. Harris. Progress on the kbsa performance estimation assistant. In *5th Annual KBSA Conference, Syracuse, NY, Sept. 1990*, 1990.

[9] R.A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Trans. Software Eng.*, se-11:32–43, 1985.

[10] P.E. London and M.S. Feather. Implementing specification freedoms. In C. Rich and R. Waters., editors, *Readings in Artificial Intelligence and Software Engineering*, pages 285–305. Morgan Kaufmann, 1986. Originally published in Science of Computer Programming, 1982 No. 2, pp 91-131.

[11] T.I.Oren M.S. Elzas and B.P. Zeigler. *Modelling and Simulation Methodology in the Artificial Intelligence Era.* Nort Holland, 1986.

[12] L.F. Pollacia. A survey of discrete event simulation and state-of-the-art discrete event languages. *Simulation Digest*, 20:8–25, 1989.

[13] H.B. Reubenstein and R.C. Waters. The requirements apprentice: Automated assistance for requirements acquisition. Submitted to the IEEE TSE, 1990.

[14] P. Zave and W. Schell. Salient features of an executable specification language and its environment. *IEEE Trans. Software Eng.*, se-12:312–325, 1986.

[15] B.P. Zeigler. *Object-Oriented Simulation with Hierarchical, Modular Models : Intelligent Agents and Endomorphic Systems.* Academic Press Inc., 1990.

Knowledge-Based Software Assistant : A Knowledge Representation Model and its Implementation

N. Boudjlida - T. Khammaci
CRIN - Nancy I University - B.P. 239
54506 - Vandoeuvre Les Nancy Cedex (France)
e-mail address : nacer@loria.crin.f

Abstract : The Knowledge-Based Software Assistant approach aims at providing support environments which play an active rather than a passive role in assisting developers. Active support can result from the utilization of knowledge about the tools which can be used, the policies to be obeyed, due to the policies induced by the tools or to those induced by the method(s) used at any step of the software life-cycle. In this paper, we first propose concepts for software process modeling in order to capture and describe all the knowledge and how to assist automatically, wherever possible, all the software partners.

Design and implementation issues are also considered. Thus, we report on UPSSA, a KBSA prototype which uses a subset of a Model for Assisted Software Process. The prototype enables triggering sequences of actions in order to help the software developer and to control the software process.

Keywords : Software engineering, Software process modeling, Software Assistance, Knowledge based System, Knowledge Based Software Assistant, Development Assistant.

1 Introduction

Significant progress has been made in providing automated assistance for software production. Work on this topic have progressed in different ways. One way is the software engineering workshops which are method or language-dependent. Another more recent way is the Computer Aided Software Engineering (CASE) [4][15][16] which should be used not only for automating and managing the development but also for acquiring, making explicit and exploiting information of strategic organizational value [7]. The final way is the explicit modeling of the Software Process [22][19][1] and the development of mechanisms to interpret a given software process model. This is the approach we have adopted in the ALF Project [1][2] (Advanced software engineering environment Logistics Framework) which aims at extending PCTE (Portable Common Tool Environment), the European Environment Standard [23], with guidance and assistance facilities. In the ALF project, assistance functions consist in controlling, taking initiatives, helping and guiding, explaining and observing. A set of concepts [9] enables generalizing assistance in order to take shape by basic mechanisms in an Environment Kernel. This paper is organized as follows : in section 2 some projects in the area of modeling and assistance in software development are reported. The third section of this paper defines assistance functions and concepts of the Model for Assisted Software Process (MASP). In section 4, we describe an algorithm which summarizes the knowledge manipulation facilities which are required in a MASP interpretation environment. Software environment deals with coarse grained data such as program compilations, software configurations, as well as fine grained data such as statements, expressions: thus. in section 5, we first present an example of such a capability. which we have considered while studying the assistance. then we outline the objectives and the architecture of the UPSSA (Using Pre-postconditions for Simulating Software Assistant) prototype we have developed to validate our proposals with regard to assistance in software development component. Section 6 concludes with some perspectives.

2 Related work

Considerable attention is devoted at the moment in the field of modeling and assisting in software development activities. Presently assistance design must appeal to progress in two areas : data representation in databases [26] and knowledge representation techniques in artificial intelligence [21]. Consequently, a number of research projects propose to use artificial intelligence in the software life-cycle [3][15][16][18].

A first approach relies on an object base and a rule base. The *object base* is the repository of all the information needed for the development of a software project. It contains the description of object-types and occurrences of objects, the description of object relationships..... Even more, it can be extended with additional information like historical information and measurement information. The *rules* specify the precondition of an activity and its postcondition. Then, they are used as the basis for reasoning during the effective software development. Mainly, they help in *forward reasoning* to help a user or the system going further or to find out the actions the environment can undertake (This is called opportunistic processing in [15] and initiative in [6]). They also serve in *backward reasoning* to determine what activities have to be done prior to an activity asked by the user but which is not possible in the current state of the development because its precondition is not satisfied. The rules can also serve in *plan generation*, that means that the system may use them to try to find out the sequence of actions that can be performed to lead the development into a given state (see [5][16][18] for more details). The interest in this approach is that the "intelligence" is outside the tools. Thus, no change is required for the existing tools or for the ones which may be incorporated into the environment. Further, the set of rules is easily modifiable since its a "by essence" a modular approach. Moreover, different kinds of rules can be defined. and more especially. one can define strategic rules (or *meta-rules*) which drive the system in reasoning.

An other approach provides predefined structures for the object and the knowledge bases which are relevant to an application domain. [24][20] are software-engineering oriented while [25] is information-system oriented.

The Rome Air Development Center Project [3] also concerns this topic. The goal of this project is to develop both a Knowledge-Based Software Assistant and the enabling supporting technologies. Four features mark the vision of KBSA system. The first is a language which provides the user with the ability to capture the formal semantics of the system under development. The second is a general inferential system which supports an efficient reasoning. The third is a domain-specific inferential system which extends the general inferential system to include aspects which are specific to software development. Finally, the fourth is the integration into a system of the assistance needed all along a software life-cycle including project management.

The ALF project has similar objectives. It aims at extending PCTE, the European Environment Standard, in the way of guidance and assistance mechanisms. The approach followed in ALF, is to provide means for expressing various kinds of knowledge relative to software development activities and relative to how we wish these activities to be performed. The ALF system will provide accurate mechanisms for expressing and exploiting this knowledge with a particular emphasis on assistance mechanisms. The assistance concerns the various *roles* (project management, project development, ...) and should cover the entire software life-cycle. Thus, it comprises the project management assistant which provides knowledge-based help to managers in resource management (human, financial), project scheduling, ..., and the software developer assistant which provides knowledge-based help in the development process itself and in its understanding and learning. The conceptual framework of the knowledge representation is called Model for Assisted Software Process (MASP) and its concepts are addressed in the next section.

From an architectural standpoint, we distinguish four main layers in the ALF system which is currently under development. The *object management layer* is realized by the object management system of EMERAUDE [10] which is an implementation of PCTE. The *information system layer* uses the services of the Object Management System for collecting, storing, retrieving and updating information which concerns the model of a Software Process and the software-projects developed in conformity with a given model. The *piloting system layer* is in charge of coordinating the activity of the ALF-system's components (MASP interpreter, various reasoning mechanisms, MASP design component, ...). Finally, the *interface layer* encompasses various users' interfaces like MASP designer's interface, Project Manager interface, Software developer interface, ...

In the following, we focus on the various kinds of assistance we have defined in the context of the ALF project. Then we briefly expose the concepts for process modeling which are used and we report on a partial implementation of these concepts.

3 Assistance and Model for Assisted Software Process

Originality in our aim about assistance in software engineering lies principally in the determination of all assistance functions and their generalization and, in qualitative improvement in order to concretize them by basic mechanisms in an environment kernel.

3.1 Basic functions of assistance

The functions of assistance are classified according to [12] and [4] :

- *To Control* : depending on the current state of the development, an activity can or cannot be performed. Thus, when a user invokes an operation to perform an activity, the system controls whether the activity is allowed or not.

- *To Take initiatives* : the system may decide on its own to perform some operations without human interactions.

- *To Help and to Guide* : a user can expect the system to help and to guide her or him in her or his work. We consider the following *facilities* as representative of the guidance function.
  - *What to do next* : this facility enables a user to understand how to continue her or his work. So, this facility must at least deliver the set of operations which are applicable in the current state of the development.
  - *How does it work* : this facility enables knowing what operation series could be performed to produce an object of a given type.
  - *How to do it* : this facility enables knowing what operations series can be performed to produce a particular instance of an object type in a particular context.
  - *What happens if* : this facility enables understanding the consequences of an operation.
- *To Explain* : explanation helps the user in understanding the software process at particular instants. For example, when an operator invokation is rejected, the system should explicit the reason why. Similarly, when the system takes an initiative, it should explain why it decided to take it.
- *To Observe* : observation provides measurement and historical information about the software development process. Thus, the software process model must give a user means to describe *what, when* and *how* to observe.

Let us now briefly and unformally visit our proposal for knowledge representation as detailed in [9][2][4]. We propose to use an *object model* for describing the variety of objects, an *operator model* for describing the tools and. *rules, characteristics and orderings* for expressing the policies which may be induced by the tools or by the method(s) used at any step of the software life-cycle, either as a development method or as a managerial method. Further, we consider three abstraction levels :

- The *MASP or model level* concerns the modeling of software-project development and organization in terms of the proposed concepts.
- At the *instantiation level*, some objects are created and some values may be fixed for parts of the description resulting from the model level.
- The *Software Process level* is closely related to the operating system processes.

3.2 Concepts for Software Process and Assistance Modeling

The aim of the *object model* is to provide a description of the object structures by means of object properties and relationships among objects. This description is based on the entity-relationship data model [8] extended with ISA-relationships and multivalued attributes. The *ISA-relationship* enables the specification of sub-types. This kind of relationship enhances the original data model and enables a more "sophisticated" description of the object model [13][14]. Moreover, it assumes the availability of inheritance mechanisms to make the sub-types of a given type T inherit attributes of T.

The object model describes the structures which should be produced or those on which activities should be performed. An abstraction of these activities is described by the *operator model*. Even operators may be typed in that sense that an operator-type (e.g. edit) defines a class of operators, while an operator (e.g. edit using Emacs or edit using ed) is considered as a member of a given class. This operator-typing mechanism eases adding new tools to the environment. An additional feature of an operator indicates whether it may be processed by the system alone, or it needs any user's cooperation. The operator is then respectively considered to be *interactive* or *non-interactive*. This information helps the system in taking the initiative to activate an operator without a user's stimulus.

Expressions describe particular states of part of the system. They enable the specification of the operators' pre-conditions and post-conditions. They are also used in the rule and characteristics components of the MASP. An expression may have two parts : a logical part which specifies properties of the objects concerned with the expression and an optional event part which specifies *when* the expression has to be evaluated.

Policies for operator's activation and temporal constraints on operators are specified as *Orderings*. They explicitly *restrict* the activatable operators with respect to already activated ones, an activatable operator being an operator whose pre-condition is true and whose activation does not contravene the Orderings.

Particular policies and system behavior can be expressed as a set of *rules* which describe system's reactions when specific situations are encountered. Each rule associates an expression (condition-part or left-hand-side of the rule) with an operator-type (action-part or right-hand-side of the rule). The action-part specifies *what* to do *when* the situation specified by the condition-part arises. They are the basis for the system's initiative.

Finally, the *characteristics* of a MASP are a set of constraints (specified as expressions) which are satisfied during the activation of this MASP. So, they may serve as integrity constraints and as a basis for reasoning mechanisms which are activated by the system when a characteristic violation is detected.

3.3 MASP Instantiation considerations

A MASP is a purely static description of a software project. At first, no object exists except the tools which implement some operators. Object creation is progressive. The values of some of them are externally provided. For instance, the values of objects like manpower, schedules, ... are provided by the project manager. Others are created during effective work. An IMASP (Instantiated MASP) is an instance of a MASP, in which there exists a set of objects which conforms to the MASP's object-model and a set of operators which conforms to the MASP's operator-model. The IMASP is identified by the name of these two sets and by the name of its associated MASP.

Thus, IMASPs are created by necessity. Indeed, a static instantiation in which the structure of MASPs is completely Instantiated before beginning the execution is not flexible enough. Intertwining instantiation and execution phases enables considering part of development that has been executed before instantiating a further part.

Moreover, an IMASP may import the object set (or part of it) from another one if the two

IMASPs issue from the same MASP. An IMASP can be deleted by its owner (typically, at the end of the MASP calling) with classical care concerning the shared components.

So, an IMASP resembles the static context of operating-system's processes. The lowest level of our model is the Assisted Software Process (ASP) which roughly corresponds to operating-system's process except that it should conform to the defined policies.

Let us now tackle the dynamic point of view considering an algorithm which shows the main functions for knowledge manipulation within a MASP interpretation environment.

4 Software developer assistance : A general algorithm

The simple version of the algorithm described below summarizes a software developer session. We assume that an IMASP exists and that an ASP is created. i.e. the user is under the control of an ASP.

The execution of an operator by a software developer can be assimilated to the notion of transaction. Thus, when a transaction is started, some actions must be carried out. If any of these actions fails, the system rolls back the transaction.

Checking if the operator is activatable (i.e. is its precondition true ?) is required first. If the operator is not activatable, the system tries to generate and then to execute a plan to fulfill the operator precondition.

Further, at the end of the operator's execution, the system checks the characteristics. If any of them is not true, the *operator postexecution-planning* will try to fulfill it. If the system cannot satisfy the characteristics using non interactive operators, the user is requested to find a way for fulfilling them. An interaction between the user and the system is needed. If after this option, the characteristics remain not satisfied, the system rolls back the transaction.

```
PROCEDURE Operator-execution (Oper, Param, Bool);
Begin
   Exam-if-oper-is-activ (Oper, Param, Bool);
   If (Bool = False)
      Then Oper-preexec-plan (Oper, Param, True, Bool);
   If (Bool = True)
      Then Begin
             Exec-oper (Oper, Param, Bool);
             If (Bool = True)
                Then Begin
                       Check-char (Bool, Viol-Char);
                       If (Bool = False)
                          Then Begin
                                 Oper-postexec-plan (Bool);
                                 Term-oper (Bool);
                               End
                     End
```

```
            End
        Else Term-oper (False);
End.
```

The *Exam-if-oper-is-activ* routine checks the operator's precondition against the object base. If it evaluates to true, the *Exec-operator* routine launches it. Otherwise, the *Oper-preexec-plan* routine is in charge of finding plans which fulfill the precondition. This routine involves a plan-generation algorithm which uses the operator's precondition as a goal. If such a plan exists and if it involves only non interactive operators, the *Plan-exec* routine sequentially executes it, enforcing so, the precondition. When the plan involves interactive operators, it is executed with the help of the user.

At the end of the operator's execution, the characteristics are expected to be true. If any of them is not, following their checking by the *Check-char* routine, the system attempts its enforcement using a strategy similar to the strategy used for preconditions enforcement. The *Oper-postexec-plan* routine tries to generate a plan whose goal is the violated characteristics and the *Plan-exec* routine executes it with or without the user's interaction. In case of failure of characteristics enforcement, the transaction is rolled back.

Notice that information about all these steps is kept. This information will serve for generating explanation.

In the next section, we show, using the example of separated-compilation in Ada, how we formalize on the one hand objects and relationships, and on the other hand operators and ordering.

5 Objectives and architecture of UPSSA

As mentioned before, our work concentrates only on assistance in software development. So, in the current version of the prototype, we restrict the MASP concepts to the object model, the operator model and the characteristics. This work concentrates on the use of pre-conditions and post-conditions, objects and relationships between them. The constraints complete the pre-conditions operator as necessary condition for its execution.

In this section, we begin considering a typical example of software components. Note that, a software environment may need to work with program versions as a unit and also may need to work with programs as a unit. A good example of the latter capability is the ability to control recompilation of programs source code based on mutual dependencies. The idea is to use dependencies to determine exactly those modules or files which need recompilation and to automatically issue the commands to do those recompilations. This capability is similar to the one provided by the Make program [11].

5.1 Example : Separate compilation in Ada program

All Ada program units generally have a similar two-parts structure, consisting in a specification and a body. The specification identifies the information visible to the user (the interface), while the body contains the unit implementation details which can be logically and textually hidden from the user.

5.1.1 Separate compilation issues

Ada enables submitting the text of a program in one or more compilations. The *compilation units* of a program are said to belong to a program library. Formally, every compilation unit is called a library unit or a secondary unit (bodies and subunits). If we have some previously compiled library units, another unit may apply a **with**-clause to gain visibility of any given unit. From that point onward, selected component notation may be used to achieve visibility of the package components, or a **use**-clause may be applied to achieve direct visibility. The reference to a library unit in a context specification identifies a *dependency among program units* which affects the order of compilation and recompilation.

Also, in Ada we may create subunits which can be compiled separately. Notice that we must include a **separate**-clause to identify the parent unit. Similarly to the with-clause, a separate clause defines a *dependency among program units*. In addition, a subunit may have its own context specification, identifying the library units it needs through a with clause.

Using the Entity-Relationship data model extended with the ISA relationship, we can represent objects and relationships of an Ada system universe as follows (figure 1) :

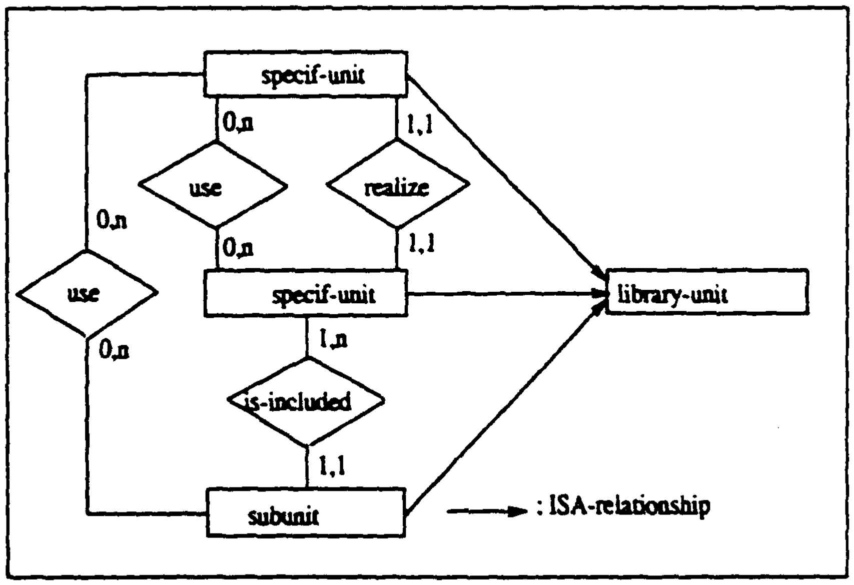


Figure 1: E-R Ada system universe

5.1.2 Ordering of compilation and recompilation

Whenever a system is built from several compilations, the dependencies explicitly defined among units require that they must be compiled in a certain order. Basically, the rule is that a given unit must be compiled before it can be visible to another unit. In particular, the specification of a subprogram, package or task must be compiled before the corresponding body.

Thus, ordering rules are constraints which must be satisfied during operators activation. Figure 2 represents the order of compilation considering an instantiation of the object model.

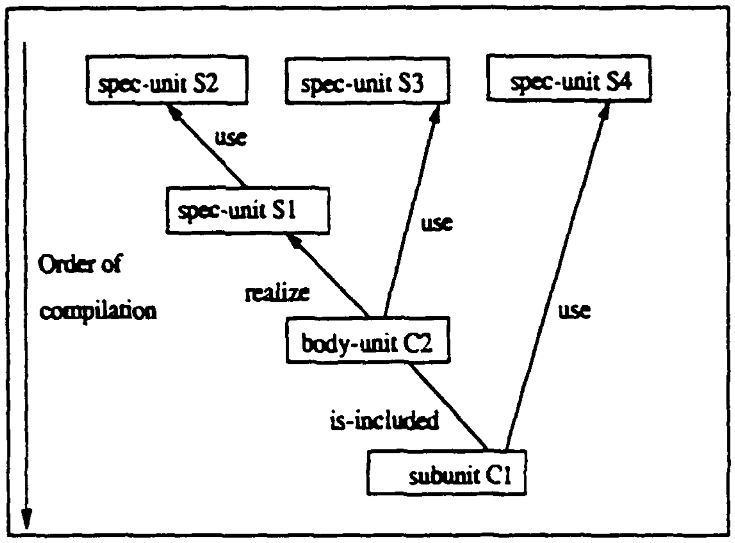


Figure 2: Ada's ordering compilation

1. The specification-unit S1 must be compiled after the specification-unit S2 is compiled.
2. The body-unit C2 must be compiled after the compilation of the specification-units S2. S3. S1.
3. The subunit C1 must be compiled after the compilation of the specification-unit S4 and the body-unit C2.

Figure 2 also shows that the constraints define a partial order (no order is imposed among the units S2. S3. S4). Any order which respects this partial order is valid. This order allows to establish the possible recompilations consecutive to the recompilation of a unit.

5.1.3 Modeling using a restricted set of the MASP concepts

Part of the previous text can be modeled as follows :

- Entity types : specification-unit. body-unit, subunit and library-unit.
- Relationship types : use, realize, is-included and isa.
- Operator types (enhanced by constraints) : compile-specif, compile-body, compile-subunit, modify-specif, modify-body and modify-subunit. We describe in the following two of the operator-types :

```
- Operator Compile-specif
 . Signature: specif-unit -> specif-compile
 . Pre-cond:  x:specif-unit and z:specif-unit and use(x,z) ==>
              s:specif-compile and is-specif-compiled-in(z,s)
 . Post-cond: y:specif-compile and is-specif-compiled-in(x,y)

- Operator Compile-subunit
 . Signature: subunit -> subunit-compile
 . Pre-cond:  x:subunit and z:body-unit and is-included(x,z)
              and t:specif-unit and use(x,t) ==>
              s:specif-compile and is-specif-compiled-in(t,s)
              and r:body-compile and is-body-compiled-in(z,r)
 . Post-cond: y:subunit-compile and is-subunit-compiled-in(x,y)
```

5.2 Objectives and architecture of UPSSA

5.2.1 Objectives

The UPSSA prototype has been designed as an integrated tool for simulating assistance in software development. The design was driven by the following objectives :

1. To constitute a knowledge base composed of software development objects and inference rules, and in particular how they can be formally represented and used for further reasoning.
2. To build an inferential system which supports reasoning and in particular, how to capture such things as logic in inference rules and data structures.
3. To ease interaction with the user by offering to her or him an easy to use interface.

5.2.2 System architecture

The search of the solution which consists in finding sequence of actions to be eventually triggered in order to control the software process, is close to plan generation in artificial intelligence [21].

Like most knowledge-based systems, our prototype is organized around a knowledge base composed of a set of rules and a set of facts. The set of rules captures the behavioral aspect of objects while the set of facts describes the objects and their relationships. Using the set of rules, the inference engine carries out the deduction process. Finally the interface addresses the end-user. In the following, we explain in more details the various components of the prototype.

1. Knowledge base : As stated before, we have two types of knowledge : facts and rules.
   - Rule base : To represent this knowledge, we use production rules. Any operator enhanced with constraints, specified in terms of the MASP operators concepts is implemented as a statement of the form : IF condition THEN action. Notice, however, that in the current version we have implemented only expressions which are in the conjunctive

form. The other expression forms (disjunction and implication) is presently in progress. Three important classes of rules have been distinguished :
The first class describes elementary operators with their pre and post conditions, their adding-list and their suppressing-list.

```
Example : Operator Modify body-unit
             . Pre-condition
                 X/body-unit and  Y/specif-unit  and  X realize Y
             . Add list
                 1- X/body-compile
                 2- X is-bu-compiled-in Z
             . Suppression list
                 ()
```

The second class includes rules of the deductive process. These are rules for chaining the elementary actions.

```
Example : Operator Compile specif-unit
             IF     X/specif-unit and Y/specif-unit and X import Y  (*)
             THEN   Compile X and X/su-compile and X-su-compiled-in Z
```

The facts X/specif-unit and Y/specif-unit are in the fact base, while X import Y is not. In this case, it is a goal for the deductive process. So, the rule below is used.

```
Deduction rule
             IF     X/specif-unit and  Y/specif-unit and  X use Y
             THEN   X import Y  (*)
               and  Compile Y  and  Y/su-compile  and  Y-su-compiled-in Z
```

The third class encompasses the rules which start up and terminate dialogues between the user and the system.

- Fact base : It contains the objects and the relationships in the system development. Each object and each relationship is represented by a predicate.

2. Inference engine : It guides the search of the rules which may be applied for a set of facts. The process combines backward and forward chaining. The proof of the goal uses backward chaining and the last rule used to prove the goal is then is applied in forward chaining in order to evaluate its actions. Also, the engine is nonmonotonic and dynamic. It uses the closed world assumption and it functions by attempts [21].

3. The interface of the prototype addresses the end-user. The end-user is in charge of the creation and the modification of the fact base. Presently, she or he is also responsible of the creation and the modification of the base of behavioral rules, but we plan to address it to an expert in software development. Figure 4 shows the interface and the corresponding processes. The prototype offers to the end-user the ACTUALIZATION, RUN and EXPLAIN functions. ACTUALIZATION enables the end-user to introduce and to update the behavioral

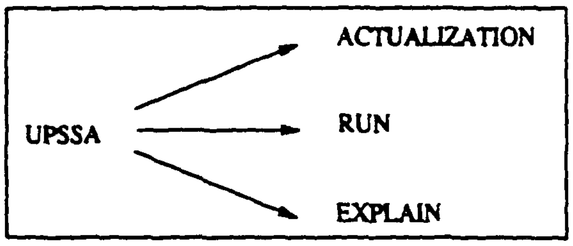


Figure 3: Interface and processes of UPSSA

rules and the facts. The RUN process yields a sequence of actions to control the software process. EXPLAIN informs the end-user about the rules used in the deductive process.

6 Concluding remarks and further research directions

We have presented in this paper, the assistance functions and a model for assisted software process (MASP) which is the basis of the ESPRIT ALF Project.

A KBSA system concerns the application of knowledge-based techniques in assisting software. Components of a KBSA system comprise the project management assistant, the requirements assistant, the specification assistant and the software development assistant. This paper reported on our approach for the Development Assistant system. The conceptual framework is a set of concepts for modeling the objects, the operations and the policies of a Software Process. We also considered some implementation issues while presenting the architecture and the objectives of the UPSSA prototype. It embodies a fact base and a rule base. Plan generation techniques are used to determine sequence of actions which may be needed for controlling the software process. The current release of the prototype is based on a subset of our modeling concepts and ensures part of the assisting functions we have identified. The implementation of the ALF system, which is currently running, deals with all the concepts and all the assistance functions for the achievement of a model-based Software Environment Kernel.

Moreover, additional work is running on the model itself. Indeed, considering the limits of the entity-relationship data model, we are experimenting an object-oriented approach for Software Process Modeling [17].

Acknowledgements : This work is partially sponsored by the Commission of the European Communities under the ESPRIT program (Project Ref. No 1520). The partners involved in the ALF project are GIE Emeraude (France), CIG-INTERSYS (Belgium), Computer Technologies Co.-CTC (Greece). Grupo de Mecánica del Vuelo, S.A. (Spain). International Computers Limited (United Kingdom), University of Nancy-CRIN (France), University of Dortmund-Informatik X (Germany), Cerilor (France), Université Catholique de Louvain (Belgium) and University of Dijon-CRID (France).

References

[1] ALF. *Advanced Software Engineering Environment Logistics Framework. Technical Annex* Project 1520. C.E.C.. June 1987.

[2] K. Benali. N. Boudjlida, F. Charoy, J.C. Derniame. C. Godart. P. Griffiths, V. Gruhn, P. Jamart. A. Legait. D.E. Oldfield. and F. Oquendo. Presentation of the ALF project. In *International Conference on System Development Environments and Factories. Berlin. RFA*, May 1989.

[3] K.M. Benner. Knowledge-Based Software Assistant. *Knowledge-Based Systems*, 1(4):221–225, September 1988.

[4] N. Boudjlida, F. Charoy, J.C. Derniame. and C. Godart. Modeling of Software Process Assistance. In *Proceedings CASE'89*. London. UK, July 1989.

[5] N. Boudjlida and O. Gervaise. Concepts and experiments in intelligently-assisted software development. Research Report, September 1988.

[6] N. Boudjlida, O. Gervaise, C. Godart, and J.C. Derniame. Fundamental Concepts and Techniques for Environment Kernels - Invited Paper. In *2nd Congress on databases - Korea Information Science Society*. Seoul, February 1988.

[7] M. Chen. J.F. Nunamaker Jr. and E.S. Weber. Computer-aided software engineering : present status and future directions. *Database Spring*, 1989.

[8] P.P. Chen. The Entity-Relationship Model: Toward an Unified View of Data. *ACM Transactions on Database Systems*. 1(1):9–36. March 1976.

[9] J.C. Derniame. H. Ayoub. K. Benali. N. Boudjlida. C. Godart. and V. Gruhn. Towards Assisted Software Processes. In *Proceedings CASE'88*. Cambridge. MA. June 1988.

[10] Gie Emeraude. The Emeraude Environment. August 1987.

[11] S.I. Feldman. MAKE : A program for Maintaining Computer Programs. *Software - Practice and Experience*. 9:255–265. April 1979.

[12] C. Godart. F. Charoy, and J.C. Derniame. Computer assisted software engineering : characterisation and modeling. In *Proceedings ICCI'89*. Toronto, Canada. May 1989.

[13] M. Hammer and D. McLeod. Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*. 6(3):351–386. September 1981.

[14] R. Hull and R. King. Semantic Database Modeling: Survey. Applications. and Research issues. *ACM Computing Surveys*. 19(3):201–260. September 1987.

[15] G.E. Kaiser and P.H. Feller. An Architecture for Intelligent Assistance in Software Development. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 180-188. Monterry. CA. 1987.

[16] G.E. Kaiser. P.H. Feller. and S.S. Popovitch. Intelligent Assistance for Software Development and Maintenance. *IEEE Soft*, 5(3):40-49. 1988.

[17] T. Khammaci and N. Boudjlida. Towards an Object-Constructor Model to Software Process Modeling. *Submitted to Information Technology Conference*. Edmonton. Canada. October 1990.

[18] T. Khammaci, N. Boudjlida. and J.C. Derniame. Knowledge-Based Software Assistant : The Case of Software Development. In *ICCI'90*. Niagara Falls. Ontario. Canada. May 1990.

[19] M.M. Lehman. Approach to a disciplined Development Process : the ISTAR Integrated Project Support Environment. In *Proceedings of Int'l Workshop on the Software Process and Software Environments*. March 1985.

[20] B. Meyer. The Software Knowledge Base. In *Proceedings of the 8th Int'l Conf. On Soft. Eng.*. London. August 1985.

[21] N.J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Co.. Palo Alto. 1980.

[22] L. Osterweil. Software Processes are Software Too. In *Proceedings of the 9th Int'l Conf. on Soft. Eng.*. pages 2-13. Monterey. CA. March 1987.

[23] PCTE+. C Functionnal Specification Issue 2. CEC ESPRIT program. July 1988.

[24] M.H. Penedo and E. Don Stucle. PMDB - A Project Master Database for Software Engineering Environments. In *Proceedings of the 8th Int'l Conf. On Soft. Eng.*. pages 150-157. London. August 1985.

[25] H. Tardieu. Contribution of the Entity Relationship Approach to object management in an information system - A tutorial. In *5th Int'l Conf. on Entity Relationship Approach*, Dijon - France. November 1986.

[26] J.D. Ullmann. *Principles of Database Systems*. Computer Science Press. Rockville Maryland. 1982.

Exploiting Metamodel Correspondences to Provide Paraphrasing Capabilities for the KBSA Concept Demonstration Project

Gerald B. Williams
Center for Strategic Technology Research
Andersen Consulting
100 South Wacker Dr.
Chicago, IL 60606
williams@andersen.com

Jay J. Myers
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292

Abstract

To provide paraphrasing capabilities for the Concept Demonstration project, the paraphraser written for the KBSA Specification Assistant project has been moved to a significantly different platform without having had to modify the original code. The success of this reuse effort has been a function of our chosen approach and some inherent features of the platforms. Instead of reconciling incompatibilities between the applications at the representation level, we chose to reconcile differences at the conceptual (or metamodel) level. The result has not only been a clean migration of the paraphrasing functionality but has also revealed significant conceptual commonalities between two specification languages that are considered to be very different. The exercise has also lead to interesting speculations about the generality of system specification concepts in general.

1. Introduction

The ability to paraphrase formal descriptions into English is a key capability of any knowledge based software assistant and consequently an important feature of the Concept Demonstration effort [3]. A paraphraser to translate specifications into English descriptions was developed for the GIST specification language [4] as part of the KBSA Specification Assistant Project at the University of Southern California/Information Sciences Institute [9, 13]. The KBSA Concept Demonstration Project however, relies on an extended version of the Refine language (ERSLa[1]) as

1. ERSLa (Extended Refine Specification Language) is an extended version of the Refine language under development at Andersen Consulting's Center for Strategic Technology Research. Much of ERSLa (pronounced Ursula) is Refine. We often reference Refine and ERSLa synonymously when describing fundamental concepts of the language.

its formal specification language. Thus, a major task within the Concept Demonstration effort is to provide paraphrasing capabilities for specifications written in ERSLa that are similar to those already developed for specifications written in Gist.

In this paper, we will discuss our efforts to date in developing such capabilities. As we will describe, this task has lead to the recognition of significant commonalities underlying two different specification representations and has illustrated significant leverages provided by these languages developed to support specification-based software development. Specifically, we will outline the architectures of two versions of the paraphraser: the ARIES[2] version and the Concept Demonstration version. We will also discuss the functionality of the paraphraser components, the strategies for installing paraphrasing capabilities in the Concept Demonstration and the reconciliation of the different specification language metamodels for the purpose of paraphrasing.

Throughout this report, we use the term language metamodel (or simply metamodel) to refer to the model that underlies a particular specification language. The language metamodel is not to be confused with models that are constructed by writing specifications in that language. Rather, the metamodel is a collection of class and attribute declarations that define the range of concepts expressible in the language. Typical metamodel class declarations include declarations for specifying types, classes, relations, attributes, functions, events and so forth. Typical attribute declarations include names, subsumption relations (e.g. subclass and superclass), declaration parameters and so on. As we use the term, the metamodel also encompasses built-in or predefined declarations of logical and mathematical operations, and standard types like integer, string, symbol, sequence, and set. In essence, the metamodel defines the language concepts available to the user of the specification language. These language concepts include the elements of the languages abstract syntax. The metamodel is the conceptual framework behind every application model expressed in specification language.

The ARIES and ERSLa metamodels are variants of entity-relationship (ER) models. "Types" in ARIES and "object-classes" in Refine correspond to entities. "Relations" in ARIES and "maps" or "attributes" in Refine correspond to relationships. Other similarities of the languages exist in their representations of control structures, expressions, statements and constraints (invariants). A primary difference of the languages that needed to be reconciled at the conceptual level was the ARIES emphasis on a relational model over the Refine's greater reliance on an object-oriented approach. Despite this seemingly fundamental difference, we found a very high degree of conceptual correspondence between the metamodels of the two languages.

2. The ARIES Paraphraser Architecture

Figure 1 graphically depicts the component architecture of the ARIES paraphraser. Operationally, a formal specification written in Gist, is parsed and represented in the ARIES knowledge base. The paraphraser then views and interprets the specification representation in terms of concepts of the ARIES metamodel and generates a feature description (FD) for each proposition to be expressed about a specification component. The Unification Generator expands and translates the feature descriptions generated by the paraphraser to produce English descriptions of the target

2. ARIES is the project name for an effort to combine and extend the capabilities of the Specification and Requirements Assistants. The ARIES paraphraser is an updated version of the Specification Assistant paraphraser.

specification component or components.

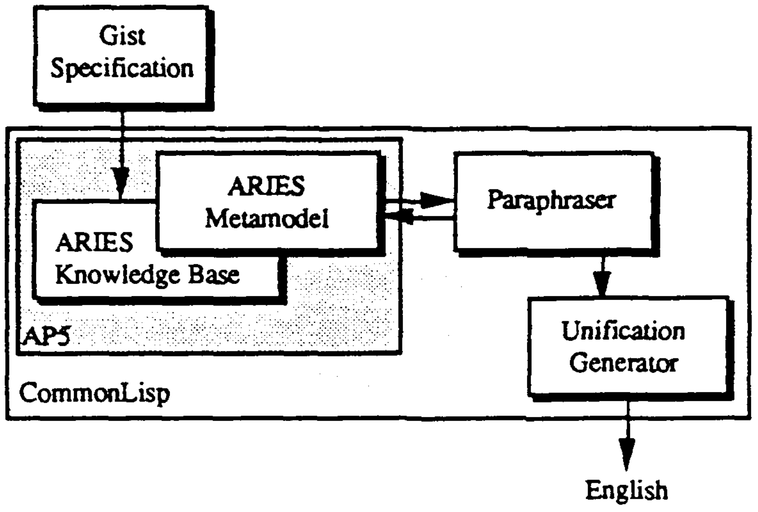


Figure 1

The ARIES knowledge base and the ARIES metamodel are built on the knowledge representation language AP5. The paraphraser accesses the knowledge base through AP5 queries but does not store or make modifications to the knowledge base. Aside from this interface to AP5 the paraphraser is independent of the AP5 implementation. The Unification Generator is completely independent of AP5. All components are build on Common Lisp. The following subsections briefly describe the individual components of this architecture.

2.1 The AP5 Platform

AP5 is an extension to Common Lisp that provides capabilities to support software prototyping [1, 2]. AP5 supports the notion of relations as programming abstractions that can be used for data storage and retrieval without commitment to a particular data structure or lower level representation. The AP5 knowledge representation language and database are built as an extension to Common Lisp. AP5 provides an interface for defining relations (i.e. data organization schemes), populating relations, accessing (i.e. querying) the representation, and modifying tuples (sequences of values) of the relations. Annotations can be attached to relations to enhance performance. These annotations provide estimates of size and cost, or specify a particular representation for a relation.

The AP5 model of computation includes an active database. It allows a programmer to define rules to maintain consistency constraints or to automatically trigger programs upon specified database updates. An extended first order logic notation is used to specify triggering conditions for rules and to formulate database queries.

Relations in AP5 can be of any arity and are either first-order, derived or computed. First order or transition relations contain only tuples which have been explicitly asserted into the relation (the contents of a database transition) and not subsequently retracted. Derived relations are defined by a well-formed formula or Lisp computation that depends on the contents of first order and/or other less derived relations. For example, a derived relation SPOUSE-AGE could be defined as the concatenation of existing relations SPOUSE and AGE using the definition:

$$\forall(x)\forall(y)\exists(z)(\ \mathrm{SPOUSE}(x,\ z) \wedge \mathrm{AGE}(z,\ y)\ )$$

Computed relations are defined by Lisp computations that do not depend on the contents of the AP5 database and whose contents are invariant. For example, a computed relation EVENP could be defined in terms of the Lisp predicate of the same name. Much of the power of AP5 comes from the ability to support derived and computed relations as opposed to having to explicitly populate each relation.

All AP5 relations are built upon a number of capabilities. Among these capabilities are the ability to test membership of tuples in the relation, to assert and retract tuples in the relation and to generate the tuples for specific projections of the relation (such as the domain values given the range values of a binary relation). These are usually supplied automatically either by default or by the choice of a particular implementation. However, AP5 permits the user to specify or extend these capabilities for a particular relation or class of relations. As we will describe, it is this feature of AP5 that allowed us to redefine the AP5 relations that comprise the ARIES metamodel in terms of semantically equivalent components represented in Refine.

2.2 The Unification Generator

The unification generator was originally developed as a part of the KBSA Specification Assistant project [8]. It is an enhanced version of the generator used by McKeown [7]. The intent was to provide a text generator that would be efficient, portable, extensible and independent of an underlying knowledge base representation. Portability was achieved by implementing the generator in Common Lisp and providing a simple application interface based on accepted linguistic principles. The paraphraser is extensible in the sense that the user can extend the lexicon with irregularities of English as needed. In applications, it has proven to be both efficient and independent of any underlying knowledge representation scheme.

The unification generator is a unification-based, surface text generator that translates simple input specifications in the form of feature descriptions (FDs) into English descriptions [6]. A feature description is a list of nested feature-value pairs which characterize the different systemic components of the concept to be expressed. The unification generator unifies the input description with a grammar represented as an extended FD. This provides defaults for unspecified features and ensures consistency among existing features and their values. The result is then translated into English.

For example, when given the simple FD:

```
((cat s)
  (process-type material-action)
  (tense past)
  (actor ((thing === programmer)
          (deictic === the)
```

```
                (number plural)))
        (process === finish)
        (goal ((thing === task)
               (deictic === the))))
```

the unification generator will produce the sentence: "The programmers finished the task." If the feature values pairs (modal could), (voice passive) and (polarity negative) were added to the above FD, the Unification Generator would produce the sentence: "The task could not be finished by the programmers.".

Some of the tasks performed by the unification generator include selecting the order of participants in the descriptions, providing for the expression of mood, tense, polarity and other features, ensuring verb/noun agreement, appropriately pluralizing nouns, conjugating verbs, handling conjoined and embedded structures, inserting punctuation, and formatting the output.

2.3 The Paraphraser

The paraphraser examines the formal descriptions represented in the knowledge base and constructs a series of FDs that are input to the unification generator. The paraphraser can be invoked upon one or more objects of the underlying representation. It identifies the class of each object and then follows a schematized procedure to construct a set of FDs that embody phrases or clauses to describe the object. The overall schema is to describe the classification of the object, retrieve and describe its major properties, and then to descend into and translate the internal structure of the object. Localized references, pronominalization and similar issues are handled within the context of these schema.

For example, if an object represents a type (or class), the paraphraser first identifies it to be a type and then retrieves its name, supertypes and subtypes among other attributes. It then identifies relations in which the type participates and descends into these. To describe relations it retrieves the name, parameters, cardinality constraints (if any) etc. It uses simple heuristics to formulate propositions to best express these facts. The resulting text for a simple type might be something like:

```
"All aircraft are mobile-objects.  Each aircraft has one
pilot.  Each aircraft is in at most one airspace."
```

In order to examine the structure of the specification objects, the paraphraser relies on a robust metamodel of the specification language. The paraphraser uses this metamodel to identify the class of the object and to determine significant properties of the object. All interactions with the knowledge base are in the form of AP5 queries using this metamodel.

2.4 The ARIES Metamodel

The ARIES metamodel consists of a hierarchy of types and binary relations defined in AP5. The types correspond to specification concepts in the ARIES representation including, for example, TYPE-DECLARATION, RELATION-DECLARATION, EVENT-DECLARATION, INVARIANT-DECLARATION, PARAMETER-OBJECT, STATEMENT, PREDICATE, REFERENCE and so forth. The relations correspond to significant attributes of the specification concepts including NAME, GENERALIZATION, PARAMETER-LIST, EVENT-DECLARATION-BODY, OBJECT-TYPE, PRECONDITION, QUANTIFIER, CARDINALITY-RESTRICTION-LIST among others.

The ARIES metamodel is derived from the grammar for Gist specifications and extended with additional relations that are input through other presentation modes (e.g.,grammatic case role annotations and relations like DATA-FLOW-LINK). The ARIES metamodel is used to organize the representation of all knowledge in the ARIES system. Various tools including the paraphraser rely upon this model in order to operate on specification objects.

3. The Concept Demonstration Paraphraser

The paraphraser for the Concept Demonstration must provide English translations of specifications written in ERSLa. The following subsections briefly describe the Refine platform, ERSLa, and the ERSLa metamodel. Section 3.4 discusses the strategies of alternative approaches considered for this project. Finally, we detail the architecture of concept demonstration paraphraser that resulted from the development strategy we selected.

3.1 The Refine Platform

Refine is a knowledge based programming environment that provides a high level programming language, a set of language processing tools (parser, compiler, etc.), interface development tools, a syntax system for extending the language, and an object-oriented knowledge base that can be queried and modified [11]. Refine provides and integrated treatment of set theory, logic, transformation rules, pattern matching and procedures. All Refine programs are represented using the knowledge base facility. This provides a powerful mechanism for examining and manipulating software objects, for example, writing transformations that can manipulate the representation of a specification.

Since Refine programs are represented using the Refine knowledge base facility, there is an explicit model of the language existent in Refine and readily accessible to the user. ERSLa extends this language metamodel. This extended model must be reconciled with the ARIES metamodel in order to use the ARIES paraphraser to generate the desired descriptions of specification components represented in the Refine language.

3.2 ERSLa and the ERSLa Metamodel

The Concept Demonstration project has extended the Refine language in ways that parallel some of the high level specification language features of the Gist specification language. In particular, ERSLa contains constructs for non-deterministic specifications, general invariants and constraints, preconditions on function declarations and will soon integrate specification constructs for representing and reasoning about time. For details of these extensions the reader is referred to [3].

The ERSLa metamodel is an extension of the Refine language model reflecting the ERSLa extensions and the influence of the ARIES metamodel concepts and attributes that are important to the paraphraser. Similar to the ARIES metamodel the Extended Refine metamodel is a hierarchy of

Refine object-classes and attributes. The object classes correspond to specification concepts in Refine/ERSLa including, for example, OBJECT-CLASS, VFUNCTION-OP, INVARIANT-OP, ANY-OP, BINDING, BINDING-REF and so forth. In addition, class definitions for specification concepts that are higher level abstractions than those that correspond to the language's abstract syntax have been created as part of the ERSLa metamodel, for example, LOGICAL-EXPRESSION, PREDICATE, CONCEPT, STATEMENT, EXPRESSION and QUERY among others.

3.3 Development Strategies for the Concept Demonstration Paraphraser

A number of options were available in order to provide a paraphraser for ERSLa specifications.

1. An ERSLa paraphraser could be constructed from scratch using the existing paraphraser for Gist as a model. This would larger be a duplication of previous efforts involving the GIST paraphraser. For the purpose of the Concept Demonstration, it is highly desirable to lever off the existing software.

2. The existing paraphraser could be modified by replacing all of the AP5 knowledge base queries involving the ARIES metamodel with corresponding Refine queries for the ERSLa metamodel (where possible). This approach was viable and less labor intensive than the previous. However, at the conclusion we would have two version of the paraphraser that would need to be separately maintained.

3. The existing paraphraser could remain intact and a correspondence model could be built in which the AP5 relations used to implement the ARIES metamodel would be redefined in terms of an extended Refine implementation. In this desirable option, we maintain only one version of the paraphraser and the correspondences. All effort then is concentrated on the correspondences and the independence of the individual software components is maintained.

By choosing the third strategy, the effort would shift from modification of the existing paraphraser to the broader problem of establishing an abstract correspondence between the ARIES and extended Refine metamodels. This was deemed to be a generally desirable objective. Furthermore, the existing paraphraser did not use the entire ARIES metamodel but identified a significant subset through its accesses. This subset was considered to be a good starting point for establishing a general correspondence between the two metamodels.

It is worth noting that with traditional software languages this latter strategy would not be possible. However, several features of the two specification languages made this latter option possible. Most important was the separation in AP5 of the abstract relational schema used to represent data from implementation concerns. The queries used to access the metamodel relations would not need to be changed even though the underlying representation was radically different. Also, we could gain significant leverage from the very flexible capabilities to derive new concepts from existing concepts in both Refine and AP5. Finally, an important enabling feature was the recognition of an already close correspondence between the modeling concepts present in Refine and in Gist specifications. The kinds of data access and the operations performed by the paraphraser were expected to be very similar despite differences in the underlying representations.

3.4 The Concept Demonstration Paraphraser Architecture

Figure 2 graphically describes the component architecture of the current version of the Concept Demonstration paraphraser. This architecture is very similar to that of the ARIES paraphraser shown in Figure 1. Here, the Refine knowledge base replaces the ARIES knowledge base and the combination of the ERSLa metamodel, the ARIES metamodel and the Correspondence model replaces the original ARIES metamodel. The Refine knowledge base and the Extended Refine Metamodel are of course built upon the Refine platform.

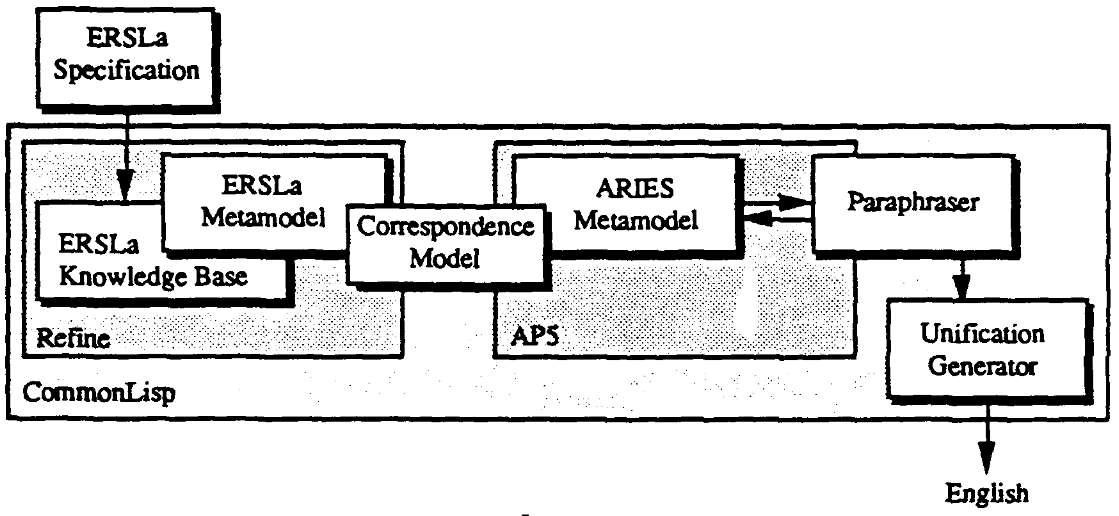


Figure 2

Operationally, a formal ERSLa specification is parsed and represented in the Refine knowledge base. The same paraphrase views and interprets the representation of a specification component through the same ARIES metamodel interface. However, the definitions of these metamodel concepts are now interpreted as Refine definitions through the correspondence model. The paraphraser and unification generator components remain as before.

4. The Correspondence Model

A correspondence is a reconciliation of concept representations from different language metamodels. A correspondence is established between conceptually similar constructs of the metamodels. The key phrase is "conceptually similar": to establish a correspondence between two concepts of different metamodels a strict isomorphism between the representations is not required. Rather, a correspondence can be constructed between different underlying representations of similar concepts independent of their underlying representations.

Because of the development strategy we chose, there is a strong sense of directionality in the correspondence model. That is, we favor the modification or extension of the ERSLa metamodel to attain agreement at the conceptual level with the ARIES metamodel. However, we should point out that the examination of each metamodel for the purpose of paraphrasing has been an inspiration for evolutionary changes in the ARIES metamodel as well as for the development of the

ERSLa metamodel.

4.1 Mechanisms for Establishing Correspondences

As described earlier, AP5 extends Common Lisp with relations that can be used to store and retrieve data using an extended first order logic query language. In most cases, relations use predefined implementations that supply their basic capabilities (i.e., testing, updating, and generating). However, an abstract interface is also available that allows a programmer to explicitly provide the capabilities to define a relation by supplying functions to test whether the relation is true for a given tuple of values, to update the data in the relation, to generate data from the relation given a specific pattern of inputs and so forth.

In developing the correspondence between the extended Refine and the ARIES metamodels, we exploited this feature of AP5 by redefining the metamodel relations referenced by the paraphraser using existing or newly defined Refine functions and maps to provide the required capabilities. Since the paraphraser does not update data or trigger actions upon changes to data, the only capabilities it needs are tests of the classification of objects and associative retrievals of binary relations (finding the range values corresponding to some value in the domain of a relation or the domain values corresponding to some value in the range of the relation). Thus, when the paraphraser attempts to test whether an object is of some type recognized in the metamodel (e.g., an invariant) we arrange for the appropriate corresponding Refine predicate to be called. When a query requires that a relation be generated (e.g., find the values of the parameters of a function call) we use a corresponding Refine map to supply the answer.

Two macros, one for types and one for binary relations in the metamodel are used to establish these metamodel correspondences. These macros define AP5 relations whose capabilities are derived from the appropriate Refine functions and maps (including functional converses for a binary relation whose domain values must be generable given values in the range). We also found it convenient to supply a default tester for supertypes consisting of a disjunction of the testers for any declared subtypes.

For a few cases, such as assignment statements, the correspondence between metamodels was simplified by adoption of the Refine concept into the ARIES metamodel.

4.2 Correspondence Taxonomy

A correspondences can be described as being one of three types: a direct correspondence, a transformed correspondence, or a derived correspondence. Derived correspondences can be further specialized into formulated and manufactured correspondences.

When conceptually similar components exist in each metamodel and the components are represented in a similar manner, then a direct correspondence can be established. By "similar manner" we mean a particular concept is common to both metamodels and it is represented as an entity in both metamodels or as a relationship in both metamodels.

The ARIES concept of an INVARIANT-DECLARATION directly corresponds to the ERSLA concept (i.e. the Refine object-class) INVARIANT-OP. Also, the ARIES relation ELSE-CLAUSE, a component of a conditional statement, directly corresponds to the ERSLA attribute of the same name. Direct correspondences require no modification of either metamodel. Establishing a direct correspondence requires little more than creating an ARIES alias for an ERSLA concept definition.

When conceptually similar components exist in each metamodel but, the components are not represented in a similar manner (i.e. not ER aligned) then it is necessary to establish a transformation correspondence that effectively equates the different representations of the concept. Perhaps the most vivid example of a transformed correspondence in this exercise is the conceptual correspondence between the way ARIES unary relations and Refine boolean valued attributes are used. In each case, the language component is used to represent some truth feature of a specification. For example the notion that an aircraft can be in-flight is represented in a Gist specification as a unary relation on aircraft. A aircraft is said to be in flight if the object representing that aircraft appears as a tuple of the unary relation in-flight. In ERSLa, the same specification feature is represented as an attribute with aircraft as the domain and boolean as the range. An aircraft is considered to be in-flight if the value of the attribute in-flight is true for that aircraft. The reconciliation of the two representations was attained by making a correspondence transformation that recognizes the binary boolean valued attributes in an ERSLa specification as unary relation in the eyes of the paraphraser.

Other metamodel concepts that have been reconciled by transform correspondences include Aires multi-valued relations and Refine set-valued attributes. In addition, the way logical and mathematical operators are represented in the ARIES metamodel differs significantly with their counterpart Refine representations. Interestingly, ARIES operators are declared as predefined relations. Refine operator declarations are object classes. This representation discrepancy is also resolved as a transformation correspondence.

When an important concept of one metamodel has no counterpart in the other, it is necessary to derive a correspondence for that concept. Frequently, the counterpart concept can be derived from an assemblage of existing representation components of the counterpart metamodel. If this is the case, a formulated correspondence is established. Often, formulated correspondences are constructed from disjunctions of concepts in the counterpart metamodel. For example, the ARIES concept TYPE-DECLARATION is interpreted to include all user specified types in addition to predefined types of the Gist language (e.g. integer, real, etc.). In Refine (and therefore ERSLA) there is an explicit differentiation in the language metamodel between the language supplied datatypes and user defined object-classes. The following predicate defines the nature of the appropriate formulated correspondence:

```
function type-declaration (x: object): boolean =
   if or(find-object-class('re::user-object) in class-superclasses(x, true),
         re::integer-op(x), re::real-op(x), re::character-op(x),
         re::symbol-op(x), re::boolean-op(x) )
   then true else false
```

The correspondence establishing mechanisms discussed in the previous section provide a convenient way of constructing formulated correspondences that can be described as disjunctions of existing metamodel components.

We find however, that not all derived correspondences can be formulated correspondences. Often the paraphraser depends on ARIES concepts that are either at a higher level of abstraction than is available in the Refine language metamodel or ones that do not exist in Refine. For example, the concept of QUANTIFIED-PREDICATE does not exist in the Refine language model and, given its ARIES semantics, is a poor candidate for a formulated correspondence. In this case , we manufacture a new object-class (QUANTIFIED-PREDICATE) as part of the ERSLA metamodel to correspond with the ARIES metamodel.

Instances of quantified-predicate are generated at run time based on the specification context relative to a representation query posed by the paraphraser. For example, in the ARIES metamodel, an invariant has an associated condition that stands in the relation PREDICATE-OF with the invariant. The value of the MONITOR-CONDITION attribute of an ERSLA INVARIANT-OP together with the implicit assumption that MONITOR-CONDITION is universally quantified, corresponds to the ARIES concept of QUANTIFIED-PREDICATE.[3] A query from the paraphraser for the PREDICATE-OF an invariant will, through the correspondence, generate an instance of the ERSLA object-class QUANTIFIED-PREDICATE annotated with information specific to the invariant being paraphrased. These annotations (i.e. attribute values) include PREDICATE-BODY, QUANTIFIER, and BOUND-VARIABLE-LIST. Note that these attributes are also products of manufactured correspondences.

5. Conclusions

Using the metamodel correspondence strategy outlined in this report, we successfully adapted an English paraphraser developed for one specification language, Gist, to another quite distinct specification language, ERSLa, thus providing paraphrasing capabilities for the Concept Demonstration. The point of interest is that the adaptation process involved no modification of the paraphrase code as it was written for the ARIES project. Instead, correspondences were established between the metamodels of the two languages. A number of factors contributed to this success:

- The Unification Generator and ARIES Paraphraser were portable, robust and largely independent of a particular specification language. The Unification Generator was by design a general, application-independent tool. The ARIES Paraphraser, while specifically designed to translate Gist specifications was shown to interact solely with a conceptual metamodel, to have a limited AP5 query interface, and to use general paraphrasing schema applicable to a broad range of specification components.

- There were numerous closely corresponding concepts in the ARIES and ERSLa metamodels. Some similarity was not surprising in light of the similar backgrounds and aims of the two specification representations. However, we were pleasantly surprised at the depth of the correspondences and the relative ease with which concepts that were only remotely similar at the level of representation could be brought into a more exact correspondence. As we have noted, on a few occasions, a correspondence was also made possible by changes to ERSLa or to ARIES.

- The AP5 and REFINE representations of the metamodel concepts proved to be extremely

3. In actuality the ARIES concept of quantified-predicate is broader than presented for this illustrative example. A complete discussion of the full correspondence would provide no additional value.

flexible and to provide leverage for our efforts to establish correspondences. AP5 provided a critical layer of separation between the paraphraser's query interface and the knowledge base representation. Furthermore, it allowed the metamodel concepts to be redefined in terms of relation capabilities specified in LISP (including Refine functions). On the other side, Refined provided the language and representation facilities for creating, transforming, and deriving correspondences from its existing metamodel.

The Concept Demonstration effort is serving as a forcing function for attempts to bring KBSA technologies into alignment. A primary task of this effort is to reconcile different approaches to the KBSA software development process. Our efforts to develop a paraphraser for the concept demonstration suggests that reconciliation of the metamodels underlying current specification languages is a highly promising path to such an end.

As an integral part of our efforts, we successfully developed a correspondence model that bridges many of the differences between the language metamodels underlying ERSLa and ARIES. As noted, these two specification metamodels were found to correspond at a deep conceptual level at least for those components amenable to paraphrasing. They use corresponding classifications of the objects found in specifications, e.g., types or object class relations or maps, procedures or functions, demonic events or rules, invariants, statements, references or bindings and so on. In addition, corresponding attributes are present in the two metamodels, e.g., names, parameters, simple count and type restrictions, quantifiers, preconditions, postconditions, and triggering conditions. Similarly, there was significant overlap in terms of control constructs and their components (like conditional statements, loops, sequential blocks) built-in operators or predefined relations (i.e., simple logical and mathematical operations, instance-type associations) and types (integral, string, symbol, set, sequence). We anticipate that in addition to its theoretical interest. This correspondence model will be useful in helping to bring into alignment other specification tools that operate in terms of a consensus specification language metamodel.

The correspondence model that has resulted from this effort can be viewed as a first step toward consensus regarding a conceptual model for functional specification using an entity-relation framework. It is important to pursue this kind of understanding of the concepts considered to be useful or essential for functionally specifying systems.

We have also seen that the core concepts present in our correspondence model are largely limited to traditional database and programming language concerns (the metamodel also supports graphical specification formalisms like data flow and system decomposition [5]). The inference we make from this observation is that the specification metamodels are for most purposes language bound. That is, they are concerned almost entirely with the representation of language specific concepts.

We envision a model to support specification that goes beyond these concerns to include higher-level concepts that readily correspond to the kinds of abstractions underlying informal, or natural language specifications. We need to be able to devise specification models in terms of general concepts of actions and propositional relations, participants, roles and circumstances as found in conceptual dependency theory [12] or the upper model of natural language systems [10]. The ARIES project has taken the first steps in this direction by including a library of general predefined specification components in its environment [5].

The tasks of paraphrasing and explaining specification constructs appear to have much to contribute by way of identifying the natural abstractions that are fundamental to a specification formalism. We contend that the language of good explanation can be used to provide a good conceptual model from which languages for specification can be modified, extended or designed.

7. References

[1] CLF Project Team, *AP5 Training Manual*, USC Information Sciences Institute, 1990.

[2] Cohen, D., *AP5 Manual*, USC Information Sciences Institute, 1990.

[3] DeBellis, M. "The KBSA Concept Demonstration Prototype", *Proceedings of the 5th Annual RADC Knowledge Based Software Assistance Conference*, Syracuse, New York, September 1990.

[4] Goldman, N., Wile, D., Feather, M., Johnson, W.L., *Gist language Description*, USC Information Sciences Institute, 1988

[5] Johnson, W.L., Harris, D.R., "Requirements Analysis Using ARIES: Themes and Examples", *Proceedings of the 5th Annual RADC Knowledge Based Software Assistance Conference*, Syracuse, New York, September 1990.

[6] Kay, M., "Functional Grammar", *Proceedings of the 5th Annual Meeting of the Berkeley Linguistics Society*, 1979, p 142-158.

[7] McKeown, K., *Generating Natural Language Text in Response to Questions about Database Structure*, Ph.D. dissertation, University of Pennsylvania, May 1982.

[8] Myers, J.J., "A Standard for KBSA Text Generation", 1989, unpublished.

[9] Myers, J.J., Johnson, W. L., "Towards Specification Explanation: Issues and Lessons", *Proceedings of the 3rd Annual RADC Knowledge Based Software Assistant Conference*, 1988, p 251-269.

[10] Penman Natural Language Generation Group, *The Penman User Guide*, USC Information Sciences Institute, 1988.

[11] Reasoning Systems Inc., *Refine User's Guide*, Palo Alto, Calf. 1990.

[12] Schank, R., *Conceptual Information Processing*, North Holland, New York, 1975.

[13] Swartout, W., "GIST English Generator", *Proceedings of the National Conference on Artificial Intelligence*, 1982.

Appendix: Example Paraphrases of ERSLa Specifications:

This appendix illustrates example paraphrases of some ERSLa specifications for the ATC domain. The paraphrases are shown here for the purpose of giving the reader some indication of the range and flexibility of the Concept Demonstration paraphrasing capabilities. These examples are partial and have been selected from results of different paraphraser modes.

ERSLa specification for the ATC domain:

```
var aircraft-spec-domain-object: object-class subtype-of upper-model-
domain
var air-location: object-class subtype-of location
var altitude: map(air-location, real)
   computed-using altitude(ac) = 30000.0
var mobile-object: object-class  subtype-of physical-object
var aircraft: object-class  subtype-of mobile-object
var aircraft-id : map(aircraft, symbol) computed-using
   aircraft-id(ac) = name(ac)
var in-flight : map(aircraft, boolean) = {||}
var aircraft-location: map(aircraft, location) = {||}
var controlled : map(aircraft, boolean) computed-using
   controlled(@@) = false
var controlling-facility: map(aircraft, faa-atc-facility) = {||}
var pilot: object-class  subtype-of person
var pilot-of : map(pilot, aircraft) = {||}
var airspace: object-class subtype-of aircraft-spec-domain-object
var airspace-region: map(airspace, region) = {||}
var airport: object-class subtype-of physical-object
var faa-atc-facility: object-class subtype-of physical-object
var controlled-airspace : map(faa-atc-facility, airspace) = {||}
var controller: object-class subtype-of person
var departure-controller: object-class subtype-of controller
var ground-controller: object-class subtype-of controller
var tower-controller: object-class subtype-of controller
var traffic-center-controller: object-class subtype-of controller
var facility-controllers: map(faa-atc-facility, set(controller))
   computed-using facility-controllers(@@) = {}
var center-facility: object-class subtype-of faa-atc-facility
var tower-facility: object-class subtype-of faa-atc-facility
var tracon-facility: object-class subtype-of faa-atc-facility
var radar: object-class  subtype-of physical-object

function take-off (ac: aircraft) =
    in-flight(ac) <- true;
    aircraft-location <- any(air-location)

invariant control-aircraft-invariant  (ac: aircraft) =
    in-flight(ac) => controlled(ac)
```

The following paraphrase is a summary of the specification structure (i.e. the entities and subsumption relations).

```
The entities in this specification are air-locations, aircraft,
   aircraft-spec-domain-objects, airports, airspaces, controllers,
   departure-controllers, faa-atc-facilities, ground-controllers,
   ground-locations, mobile-objects, pilots, radars, tower-
   controllers, tower-facilities, tracon-facilities, traffic-center-
   controllers and warnings.
Airports, airspaces and warnings are all aircraft-spec-domain-objects.
Traffic-center-controllers, tower-controllers, ground-controllers and
   departure-controllers are all controllers.
Tracon-facilities, tower-facilities and center-facilities are all faa-
   atc-facilities.
Pilots and controllers are people.
All aircraft are mobile-objects.
All radars are physical objects
```

The next paraphrase is focused on the aircraft entity, its attributes and the relations it which it participates.

```
Aircraft is the name of an object class.  There can be any number of
aircraft objects.  Each aircraft has one aircraft-location and one
aircraft-id which is a symbol.  Each aircraft can be in-flight and can
be controlled.  Each aircraft has one controlling-facility which is an
faa-atc-facility.  Each aircraft is viewed-on one radar.
```

The following is a paraphrase of the function TAKE-OFF. (Note the paraphrase of the non-deterministic ERSLa operator.)

```
An aircraft AC can take off. To perform a take-off, the system
sequentially does the following:
  1. The system sets the value of in-flight of AC to true.
  2. The system sets the value of aircraft-location of AC to any
       air-location.
```

Finally, we present a paraphrase of the above ERSLa invariant definition.

```
Control-Aircraft-Invarinat is the name of an invariant. It asserts that
for every aircraft AC, when AC is in-flight, AC must be controlled.
```

A Planning System for the Intelligent Testing of Secure Software

Deborah Frincke, Myla Archer, Karl Levitt

Division of Computer Science, University of California, Davis, CA 95616

Abstract. We present a planning-based approach to automating the use of testing techniques to detect flaws in software, which we call *intelligent testing.* In contrast to verification, which is used to show that a system satisfies a specification, our approach involves finding sample inputs that demonstrate the incorrectness of a system. We see intelligent testing as a middle step in the debugging of a system. First, the obvious bugs would be detected by desk checking or conventional testing. Then, our intelligent-testing approach would be used to identify less obvious bugs. Finally, formal verification would be used on a system which is strongly believed to be free of bugs.

Our prototype tool, TPLAN, represents the operations of the system being tested as STRIPS-like rules. The operations can represent the system at any level of abstraction, ranging from specifications of the system interface operations to statements in a programming language. We have used TPLAN primarily to identify security flaws in simple operating systems with multilevel security, the goal being to determine sequences of operations that reveal security flaws. The systems are represented abstractly in terms of *formal top-level specifications.* TPLAN can handle the following kinds of information from the user: expressions capturing a security flaw; operations likely to be part of an operation sequence revealing the flaw; skeleton sequences that are likely to be part of some security-flaw-revealing sequence.

1 Introduction

The research described in this paper addresses the issue of *testing.* In contrast to verification, which seeks to prove that a program is correct in general, testing verifies that a program works for one or more specific inputs. Typically, testing is easier to perform than verification, and can provide insight as to how to structure a formal verification of the system. Testing has an additional advantage over verification in that a failed test exhibits an actual flaw, rather than providing an unproven theorem (as is the case with most verification systems). However, it is often impossible to perform sufficient tests to show that a program works correctly in all cases[BBFM82].

The testing methodology described in this paper operates on specifications, and uses classical Artificial Intelligence planning techniques [Nil80] to develop tests to detect security flaws. A prototype system, TPLAN, has been specialized to achieve goals concerned with information flow in secure systems. Insecure systems can allow flow of two kinds between processes: *overt* and *covert.* Overt flow involves copying of information from one process to another. Covert flow involves indirect transmission of information through shared objects. Using operation specifications and a description of a particular type of information flow, TPLAN attempts to find a sequence of operations (a plan) that produces the specified flow. Information flow is described by exhibiting an initial—valid—state and a final state wherein the user has access to unauthorized information.

TPLAN produces a flaw-illustrating plan if one exists. This is in contrast to mechanical flow analyzers[Fie80], as not all the channels these systems identify actually permit information flow [Fra83], and checking them all can be time consuming and wasteful of resources. Use of TPLAN permits the user to focus on actual flaws in the system at the specification stage.

A side benefit of the planning approach is that it comes with generic flaw-revealing tests, that is, expressions that subsume all cases of input values that cause the flaw to be revealed. For a class of covert channels, an approximation to the bandwidth can be derived using an appropriate expression.

2 Background

2.1 Security

An environment's *security policy* defines ways in which subjects (human users or programs) and objects can interact within it. The security policy of a system determines both *access control* (which subjects can directly manipulate which objects) and *information flow* (what can be done with the information in objects) [Den82]. Permitted interactions are determined from the relative *security levels* of the users and objects. Security levels may be assigned either statically or dynamically, depending upon the specific environment. One commonly used policy is the *Multilevel Security* policy (MLS)[Def85]. When MLS is enforced, subjects having a higher security level can obtain information from objects having a lower security level, but can not place any there.

Information is passed between subjects via *channels* that may be either direct or indirect. *Channels* have been described by Millen [Mil87] as systems having one input and one output. An example of a direct channel is a memory location that is shared by two users. Anything placed in this location by one user is directly accessible to the other user. An example of an indirect channel is an operating system's list of available primary memory blocks. If one user 'grabs' all of the available blocks right before another user requests a block, the second user receives an error message, indirectly obtaining information about the activity of the first user.

If the channel is *noisy*, then the output for a given input has a probability distribution rather than a single value. *Covert* channels consist of information flow that is not explicit.

A typical noisy covert channel: One user sends a 1-bit piece of covert information to a second user by modifying the page fault rate of the system—a low rate indicating binary 0, a high rate indicating binary 1. This channel is noisy since other users may also manipulate the system page fault rate. Covert channels of this type–and their bandwidth—have been discussed extensively by several authors [Den82][Mil87][TGC87][McC88].

2.2 Planning Techniques

In [BF82], Feigenbaum and Barr describe a plan as "a representation of a course of action." In this work, we use planning techniques to develop a course of action (sequence of instructions) that will transfer information from one user to another. If this information transfer is illegal with respect to the desired security policy of the system, then the plan illustrates a security flaw in the system.

[BF82] describes four different approaches to planning: nonhierarchical, hierarchical, script-based, and opportunistic. We have found two of these approaches to be particularly useful in developing plans that expose security flaws: nonhierarchical and script-based. Nonhierarchical planners find a sequence of actions that achieve a collection of goals. Script-based planners use 'skeleton' plans that outline a solution to a problem or a class of problems, and then 'fill in the blanks' with actions that achieve the outlines steps.

Two commonly referenced nonhierarchical planners are STRIPS[FN71] and GPS[NS72]. Conceptually, STRIPS operates by having a table of operators along with their preconditions and effects. STRIPS compares a starting state and a goal state for differences, and repeatedly looks for a operator whose effect achieves the desired state change. GPS works similarly, in the domain of logic problems, by choosing operations that maximize difference reductions at each step.

Perhaps the most famous skeleton-refinement planner is MOLGEN [Fri79], the molecular genetics planning system. MOLGEN starts with a specific goal, selects a skeleton plan used to solve a related experimental goal, and then refines the plan to apply to the new situation.

Recently, work has been done by Linden [Lin89] on the use of planning techniques in software design. Linden considers software specifications to be partially developed plans. Our work differs, as his goal is to develop software, while ours is to test software specifications for specific flaws.

The trouble with automated testing is that it is hard to develop test cases that will exhibit all flaws. Exhaustive techniques are expensive computationally, and selective testing does not necessarily expose all flaws. There are heuristics for selecting certain test cases based on syntax—such as selecting loop boundaries, zeros, very high or low values—but these will not necessarily help to detect security flaws, which are not usually syntactic. This work is aimed towards modelling a 'standard' security flaw, and developing a test for that flaw. This flaw detection is performed at the specification level.

3 A prototype of the technique

While developing an application, a designer may wish to determine whether the system specification allows a particular type of information flow. TPLAN has been developed to detect such flows. It is implemented in Prolog, but uses its own code to focus searching. Based on an extension of classical Artificial Intelligence planning techniques [Nil80][BF82], TPLAN searches for a sequence of operations that effect a particular information flow (given specifications for the operations of a system).

To use TPLAN, the user first defines a valid state satisfying all system security constraints. Next, the user defines a state in which the suspected security flow has occurred, showing that a user now has information that was inaccessible initially. Finally, TPLAN attempts to produce a plan, i.e., a sequence of operations, that will produce the described information transfer.

Current work on TPLAN has been aimed at showing its usefulness in finding security flaws of operating systems, though the underlying methodology has applications in other areas. The next section addresses planning as it applies to detecting a sequence of operations that exhibits disallowed flows, along with a detailed description of TPLAN.

Information flow can be described by an initial and a final (or goal) state,[1] where some process has access to information in the goal state it lacked in the initial state. Thus, the transition between states defines information flow. Consider a pair of states that are identical except for the contents of some register k, to which a process P_i has read access. Further, suppose that the new contents of register k were originally accessible only to a second process P_j. Thus, the two states describe information flow from P_j to P_i. If the security policy of the system should have prohibited P_i access to this information, then the flow is insecure. TPLAN is used to determine whether or not this type of information flow can occur, by developing a plan that achieves two types of goals: differences determined by comparing initial and goal states, and preconditions of operations that TPLAN conjectures will become part of the plan.

4 Detailed description

TPLAN was implemented in Prolog in order to take advantage of backtracking and resolution. Use of backtracking provides a method whereby all possible plans may be considered,

[1]Here, *state* refers to the state of the operating system, e.g. the contents of memory and registers.

and use of resolution permits planning variables to be left unbound as long as possible. TPLAN contains four types of rules: *planning rules*, *difference computation rules*, *operation rules*, and *architecture rules*.

4.1 Planning rules

Planning rules are used to manipulate plans. There are two types: those that eliminate 'unnecessary' goals, and those that select the subgoal to be achieved next. Unnecessary goals are goals already achieved (as a side effect of solving other goals), or goals that do not cause any real change in state. For example, the system may have two goals: causing user A's register i ($R(A, i)$) to contain value X (Goal 1), and causing user A to become the currently active user (Goal 2). Suppose that in the initial state, user B is active, user A is blocked, and only active processes can modify their registers. Further suppose that the system chooses to work on Goal 1 first, and achieves it vi: the following plan:

1. Make user A active (Subgoal 1)
2. Write X into $R(A, i)$ (Subgoal 2)

Step 1 also achieves Goal 2. Steps 1 and 2 accomplish *subgoals* 1 and 2 of Goal 1.

The *second* type of planning rule determines the goal TPLAN will try to achieve first. In theory, the goals are achievable in any order: if TPLAN determines that it is impossible to achieve all gr ls following a particular order, it backtracks and tries them in a different order. Howeve , this is not always successful, since TPLAN does not recognize infinite loops in planning sequences. TPLAN may attempt to achieve a sequence of goals where the solution to the first goal 'undoes' the solution to the last goal. This is illustrated by the following example: Consider a system containing Goal 1 (described earlier) and Goal 2′: make user B active. This time, user A is active initially. The following sequence will loop infinitely:

1. Make user B active (to achieve Goal 2′).
2. Make user A active (to achieve Subgoal 1; this unfortunately undoes Goal 2′).
3. Make user B active (to achieve Goal 2′)

Both subgoals could have been achieved if TPLAN had completed both of the Goal 1 subgoals before attempting to achieve Goal 2′. To avoid this type of looping, TPLAN uses two heuristics: complete all the subgoals of a goal at one time, and complete the most complicated goals first, since these are the goals most likely to undo other goals. Rule complexity is measured by counting the number of subgoals it contains. These two heuristics are insufficient to prevent all infinite loops, so future implementations of TPLAN will contain some form of loop detection and escape.

4.2 Difference rules

Difference rules are used to detect the differences between an initial and goal state, and to set up as subgoals the elimination of these differences. For example, if process A's register $R(A, i)$ contains the value X in the initial state and the value Y in the final state, then TPLAN adds the subgoal *register difference* [[A, i, X], [A, i, Y]] to the goal list. Every state component has its own collection of difference rules, since these depend upon the component's representation.

4.3 Operation rules

Operation rules embody the semantics of system operations. Each system operation is described in terms of preconditions and postconditions. For example, a precondition of writing X into a process' register *R* is the requirement that the process be active, and a postcondition is the requirement that register R contain X.

Operation rules are not expressed directly in Prolog, but rather via a high-level description, as in Figure 1, where we specify an operation *Fetch*. *Fetch* reads a value from a specified memory location and places it into a local register. It is called with four arguments: the user doing the fetching, the register into which the new value will be written, the memory block's virtual address, and the physical offset within the block. The first **with** clause describes the system state that results. The original system state is [*Memory, Register, Mmu, CurrentUid, WaitingUid*]; thus, *Fetch* modifies only the *Register* portion of the system state.

The **produces** clause describes the result of the operation; e.g., the value of register Rn for User will be changed from its previous value (*DontCare*) to the new value (*New*). *DontCare* indicates that the original contents of Rn do not affect the result of *Fetch*. The **by** clause states that this modification is produced by the architecture rule *doStoreReg*; this rule is the one that TPLAN uses to modify its internal version of state. Architecture rules are described in more detail in a later section.

The second **with** clause introduces the preconditions that must hold before *Fetch* may actually be applied. In this example, the three preconditions state that (1) there is a physical memory location that contains the value New, (2) User has access to that location; i.e., User has a virtual address corresponding to the physical block address, and (3) User is currently active; i.e., the user id of the currently active user—CurrentUid—is identical to User. If any of these preconditions does not hold at a stage in the plan where TPLAN wishes to apply *Fetch*, then it will make them subgoals and attempt to achieve them first. If these subgoals cannot be achieved, then TPLAN cannot add *Fetch* to the plan.

```
operation  Fetch(User, Rn, Mvirtual, Mn)  with
state   Memory, RegisterOut, Mmu, CurrentUid, WaitingUid
produces  Register: [[ User, Rn, DontCare], [ User, Rn, New]]
        by doStoreReg( Register, User, Rn, Newval, RegisterOut)
with
    precond  fetchMemory(Memory, Mphys, Mn, New)
                trigger Memory
                difference [[Mphys, Mn, undefined], [Mphys, Mn, Newval]]
              end ;
    precond  memMap(Mmu, User, Mvirtual, Mphys)
                trigger Uid
                difference [[Uid], [WaitingUid]]
              end ;
    precond  equal(CurrentUid, User)
                trigger Uid
                difference [[Uid], [WaitingUid]]
              end
```

Figure 1: High-level description of Fetch

Trigger and difference clauses indicate ways in which preconditions that they are associated with can be satisfied if they are not currently true. The trigger clause specifies the particular state component to be modified,while difference clauses describe the difference to be achieved. For example, if there is a memory location containing the desired information, but the current user cannot access it (i.e., *memMap* fails), then TPLAN triggers a change in current user *Uid*.

Operation rules are used to create steps in the plan. As mentioned earlier, TPLAN tries to achieve goals that are stated as differences between states, starting from the final state and working backward to satisfy the initial state. Working on a particular goal, TPLAN searches through the system operations until it finds one that has a postcondition containing the desired difference. In our example, it would select *Fetch* to achieve the goal of modifying a register's contents. TPLAN must ensure that all an operation preconditions hold at the point in the plan where it is to be used. These preconditions are then subgoals. Note that preconditions contain architecture rules not operation rules.

Complications arise when an operation has more than one postcondition, since all of the postconditions of an operation must hold at the stage in the plan following the operation's application. If one of the operation's postconditions does not hold, it is necessary to insert a *subplan* between these stages. The initial state of the subplan corresponds to the state that holds after the current operation is applied (i.e., operation postconditions hold), and the goal state is the one to which the operation is to be prepended.

Plan	Preconditions	Postconditions
$Purge(b)$	$Bac(b) = 0; \neg Bdf(b)$ mode = privileged	$Bal'(b) = \text{syshi}; \neg Bap'(b)$ $\forall i Mem'(b, i) = 0;$ mode = unpriv
$Raise(b)$	$\neg Bap(b)$ mode = privileged	$Bal'(b) = Pal(Cp); Bap'(b)$ mode = unpriv
$Get(b, n)$	$Bap(b); Bac(b) = 0$ $Bal(b) = Pal(Cp)$ mode = privileged	$Bac'(b) = 1$ $Bac'(Mar(n)) = 0$ $Mar'(n) = b$; mode = unpriv
$Swap$	mode = privileged	$Cp' = (Cp + 1)\text{mod}P; \forall i R'(i) = SR(Cp', i)$ $\forall i SR'(Cp, i) = R(i); \forall i SMar'(Cp, i) = Mar(i)$ $\forall i Mar'(i) = Smar(Cp', i);$ mode = unpriv
$Read(i)$	mode = privileged	$R'(i) = X;$ mode = unpriv
$Write(i)$	mode = privileged	$X' = R'(i);$ mode = unpriv
$Fetch(i, j, k)$	mode = unpriv	$R'(i) = Mem(Mar(j), k)$
$Store(i, j, k)$	mode = unpriv	$Mem'(Mar(j), k) = R(i)$
$Compute_k(i)$	mode = unpriv	$R'(k) = f_k(\cup R(i))$
$Trap$	mode = unpriv	mode = privileged

Figure 2: Operations in Millen's simple operating system

4.4 Architecture rules

Architecture rules modify TPLAN's view of the system state. They may also be considered predicates to be instantiated or revoked depending upon the operation applied. For example, in an operating system having registers, there exist architecture rules allowing TPLAN to observe and modify register values within the current state. Alternately, one may consider the system to contain predicates such as "Register I of user A has value X" and "Register I of user A is modified to contain value Y." Architecture rules are used within operation rules to describe preconditions and postconditions.

5 Examples

The security flaws described in the following sections are based on those described by Millen in [Mil79]. Millen's simple operating system contains the operations shown in Figure 2, along with a block-organized memory, a memory management unit, multiple users, and a complete set of private registers for each user.

5.1 An obvious example of insecure flow

This section describes the path that TPLAN follows to come up with a very simple example of information flow. The operations available to TPLAN include those shown in Figure 2, and two more added to induce flow: *ReadX, WriteX*. System register $\mathcal{X}$ is read and written using these new operations. Clearly, flow between users may occur via this special register. As a simplification, the system is assumed to contain only two users, each with exclusive access to two blocks of memory.

TPLAN begins with the goal of finding a plan whereby one user obtains information originally contained in the second user's memory. This is stated by defining an initial state where $User_1$'s block does not contain $User_2$'s information, and a goal state where $User_1$'s block does contain $User_2$'s information.

Initial state:

$$\left(\begin{array}{lll} R(User1,i) = R_1 & R(User2,i) = R_2 & Mem(0,k) = M_0 \\ Mmu(User1,m) = 0 & Mmu(User2,j) = 1 & Mem(1,k) = Secret \\ \mathcal{X} = DontCare & & \\ \text{Current user} = 2 & & \end{array}\right)$$

Final state:

$$\left(\begin{array}{lll} R(User1,i) = Secret & R(User2,i) = R_2 & Mem(0,k) = M_0 \\ Mmu(User1,m) = 0 & Mmu(User2,j) = 1 & Mem(1,k) = Secret \\ \mathcal{X} = DontCare & & \\ \text{Current user} = 1 & & \end{array}\right)$$

1. $Mem(Mmu(User_1,m),k)_0 \neq Mem(Mmu(User_2,j),k)_0 \neq 0$
   $Mem(Mmu(User_1,m),k)_t = Mem(Mmu(User_2,j),k)_t \neq 0$
   Using $User_2$ as the initially active process and $User_1$ as the active process in the goal state shortens the plan, though this is not required for a correct plan.
   Possible Plans for Goal 1:
   - $Store(i,m,k)$ by $User_1$, with $R(User_1,i) = Mem(Mmu(User_2,j),k)_0$
   - $Purge(Mmu(User_1,m))$

   Both modify memory; however, *Purge* can only write 0.
   Choose plan *Store(i,m,k)*.

$$\overbrace{\ldots\ Store(i,m,k)}^{User_1}$$

This sets up a new goal:

2. $R(User_1,i) = Mem(Mmu(User_2,j),k)_0$ Possible Plans for Goal 2:
   - $Fetch(i,j,k)$ with t=0
   - $ReadX(i)$ with $\mathcal{X} = Mem(Mmu(User_2,j),k)_0$

If *Fetch* is chosen, TPLAN must satisfy the precondition that some register within $User_1$'s register set contains the value in $User_2$'s block. For illustration, *ReadX* will be chosen.

Choose plan *ReadX(i)*.

$$\overbrace{\ldots\ ReadX(i)\ \ Store(i,m,k)}^{User_1}$$

This sets up a new goal:

3. $\mathcal{X} = Mem(Mmu(User_2, j), k)_0$ Only a *WriteX* can cause $\mathcal{X}$ to contain the desired value. The current process cannot do the write, since it does not have $User_2$'s information. Thus, $User_2$ must have done the write into $\mathcal{X}$ earlier. This sets up a subgoal that must be achieved before goal 3.

4. Current process = $User_2$

   Possible Plans for Goal 4:

   - *Swap*

   Choose plan *Swap* and propagate the goal 3.

$$\overbrace{\ldots\ Swap}^{User_2}\overbrace{ReadX(i)\ \ Store(i,m,k)}^{User_1}$$

   A plan for goal 3 may now be be applied. Possible Plans for Goal 3:

   - *WriteX*

   Choose plan *WriteX(i)*. The plan becomes:

$$\overbrace{\ldots\ WriteX(i)\ \ Swap}^{User_2}\overbrace{ReadX(i)\ \ Store(i,m,k)}^{User_1}$$

   This sets up the final goal:

5. $R'(User_2, i) = Mem(Mmu(j), k)_0$

   This may be achieved directly by plan *Fetch(i, j, k)*.

Thus, the resultant plan is:

$$\overbrace{Fetch(i,j,k)\ \ WriteX(i)\ \ Swap}^{User_2}\overbrace{ReadX(i)\ \ Store(i,m,k)}^{User_1}$$

5.2 A more subtle example of insecure flow

The previous example described an obvious example of flow that was easily discovered by TPLAN. In that example, information was directly transferred from one user to another. This section describes a more complex example of information flow, in which information is transferred indirectly. Here, one user will observe one of two possible results, depending upon the actions of a second user.

Some new notation is introduced at this point. Object and operation names may be subscripted with plan stages: $R_t(U,i), Store_t(i,j,k)$. Plan goals such as $=, \neq$ may also be subscripted with a user level to indicate that they may only be satisfied by a user operating at that particular level. For example, $R_t(U,i) \neq_y R_0(U,i)$ indicates that user U's register i at plan stage t must not have the same value that it did at plan stage 0; further, the change in value must have been caused by a user operating at level y.

The following describes the reasoning TPLAN uses to locate such a security flaw. The relevant part of the system state is shown in this format:

$$\begin{pmatrix} Mmu(X,j) = b_1 & Mmu(Y,j) = b_2 & Mem(b_1,k) = M \\ R(X,i) = R_1 & R(Y,i) = R_2 & \end{pmatrix}$$

1. $R_t(X,i) \neq_y R_0(X,i)$

   The initial goal is to find a plan where user X gets differing values in its registers based on an action of user Y. Only user X can modify $R_t(X,i)$. Possible Plans for Goal 1:

   - $Fetch(i,j,k)$ by user X

   Thus, goal 1 will be achieved when the following goal is achieved:

2. $Mem_{t-1}(Mmu(X,j),i) \neq_y R_0(X,i)$

   We must ensure that something other than $R_0(X,i)$ is written into block b_1, where $b_1 = Mmu_{t-1}(X,j)$. This inequality is restricted to be caused by user Y, so user Y must have had access to that block at some stage in the plan. The current plan looks like this:

$$\overbrace{\ldots Swap}^{User_X}\overbrace{\ldots Swap}^{User_Y}\overbrace{\ldots Fetch(i,j,k)}^{User_X}$$

   User Y cannot modify memory unless it has a pointer to it in its Mmu. Both users cannot point to memory simultaneously, so user X must have regained access to the memory block after Y. This sets up a new goal:

3. $Mmu_{t-2}(X, j) = b_1$

   Possible Plans for Goal 3:

   - $Get(b_1, j)$

   This operation's preconditions require block b_1 to be active, have no other user accessing it, and have the proper security level, leading to three new goals:

4. b_1 is active
5. b_1 is unused
6. b_1 has level X

   Possible Plans for Goal 6:
   - $Raise(b_1)$

   This plan achieves goal 4 as well. Remaining goals: 2, 5. Possible Plans for Goal 2:

   - $Purge(b_1)$

   This portion of the plan should take place before user Y performs *Swap*, since user X no longer has access to b_1. The current plan and system state at $t-4$ are:

$$\overbrace{\ldots Swap}^{User_X} \overbrace{\ldots Purge_{t-4}(b_1)\ \ Swap}^{User_Y} \overbrace{\ldots Raise(j)\ \ Get(b_1, j)\ \ Fetch(i, j, k)}^{User_X}$$

$$\left(\begin{array}{lll} Mmu(X, j) = b_n & Mmu(Y, j) = b_1 & Mem(b_1, k) = 0 \\ R(X, i) = R_1 & R(Y, i) = R_2 & \end{array}\right)$$

Goal 2 has been achieved, since b_1 contains 0 after *Purge*. Preconditions of *Purge* must be achieved as for goal 5. This goal must be achieved as a postcondition of *Get*, since that is the only operation that makes blocks available to the system. Additionally, since user X originally had b_1, this plan is placed before user X's first *Swap*. Assuming User X initializes $R(i)$ to something other than 0, the plan and initial system state become:

$$\overbrace{\text{Initialize } Get(b_3, j)\ \ Swap}^{User_X} \overbrace{Purge_{t-4}(b_1)\ \ Swap}^{User_Y} \overbrace{Raise(j)\ \ Get(b_1, j)\ \ Fetch(i, j, k)}^{User_X}$$

$$\left(\begin{array}{lll} Mmu(X, j) = b_1 & Mmu(Y, j) = b_n & Mem(b_1, k) = X \neq 0 \\ R(X, i) = R_1 & R(Y, i) = R_2 & \end{array}\right)$$

This plan assumes that the system is in collusion with the users, since users cannot directly name blocks or force the system to obtain a particular block.

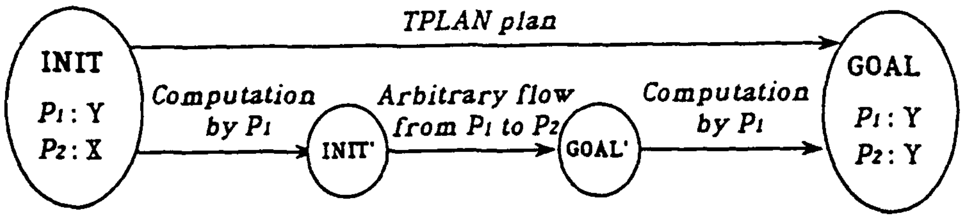


Figure 3: Generic test for overt flow

6 Uniform Tests for Information Flow

To simplify the search for plans yielding overt or covert information flow, we have considered ways to standardize the input to TPLAN.

Figure 3 shows two states, INIT and GOAL, whose essential relation is represented by encoding part of the memory content of P_1 and P_2 using variables X and Y, assumed to take values in the set 0, 1. All other state variables are distinct.

We claim that if any overt flow from P_1 to P_2 exists, then there is a plan for getting from INIT to GOAL that, treating all variables as global, accomplishes the indicated change. In fact, as indicated in Figure 3, such a plan can be obtained from an arbitrary plan for overt flow by adding an appropriate initial computation by P_1 to the beginning (to encode the value of Y) and a computation by P_2 at the end (to decode the value of Y). Thus, it is sufficient to ask TPLAN to search only for plans from a generic INIT state to a generic GOAL state.

For covert channels, we do not have a single generic test, but a family of tests that try to determine whether a process P_1 can communicate to a process P_2 by affecting something P_2 can observe via at most a single system call. The general form of such a test is illustrated in Figure 4. For each test, TPLAN is asked to find two plans filling in the gap shown, respectively ensuring that the value "?" computed from the state variables by P_2 is 0 or 1. The covert channel in the Millen example fits this model, the actions of P_1 being to release a block or not, and the single system call by P_2 being a request for a new block.

Computation of the bandwidth of such a covert channel requires knowledge of the probabilities $Pr(Sys_0)$ and $Pr(Sys_1)$ that the system state, at any given time, is such that P_2 will compute the value 0 or 1. If R_0 and R_1 are the events of receiving a 0 or a 1 and S_0 and S_1 are the events of sending a 0 or a 1, then $Pr(Sys_0)$ and $Pr(Sys_1)$ will enter into the computation of the conditional probabilities $Pr(R_0|S_0)$ and $Pr(R_1|S_1)$ in the following expression for the fractional bit of information that can be conveyed by the channel:

$$Pr(R_1|S_1) \times Pr(S_1) + Pr(R_0|S_0) \times Pr(S_0).$$

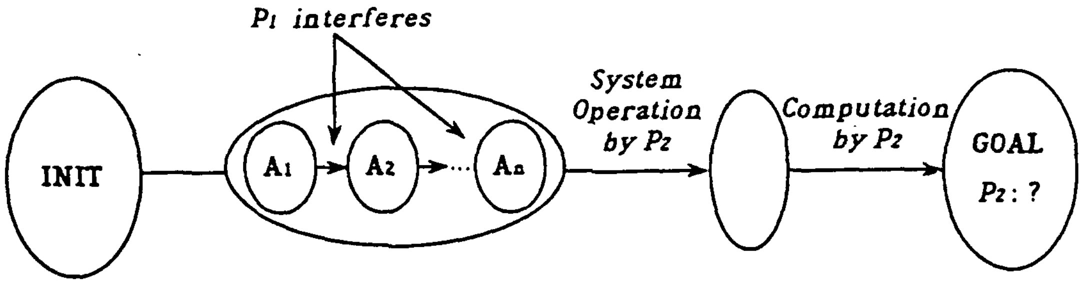


Figure 4: Standardized tests for covert flow

7 Conclusions and Future Work

Our methodology has the advantage of permitting early testing of specifications without requiring that they be executable. So far, we have applied it only to the detection of security flaws: in addition to interprocess flows in Millen's simple operating system, we have used our methodology to detect flow within a Low Water Mark system having partially ordered security levels. TPLAN is an improvement on STRIPS, which essentially does an exhaustive search, since TPLAN is guided by plan heuristics. The organization of secure systems makes them especially amenable to this type of search.

There are, of course, some limitations to TPLAN. If it terminates successfully without a plan, then we have 'verified' that the described flaw is not present. Although we can add a loop detection facility, we are not guaranteed termination, and thus cannot rely on TPLAN as a verification system.

Currently, we are attempting to use TPLAN in applications other than security. One such application involves testing a scheduler for deadlock. In this case, the goal state reflects a circular wait in resource graph of allocations and requests of resources to processes.

We are also looking at ways to extend our methodology. We wish to be able to differentiate between user level commands and system level commands, in order to determine whether a sequence of system commands demonstrating a flaw could ever arise as the result of a sequence of user commands. Reusable planning techniques, such as saving goals along with plans, may permit development of a test suite of flaws that may be applied to new operating systems. Constraints, skeleton plans, and hierarchical techniques can be used to greater advantage.

A further extension we are studying would allow the *recursive construction of infinite* plans. This would be needed, for example, in demonstrating the absence of fairness in a system by discovering a plan consisting of steps designed to preserve the possibility of denying service to some particular process at each step.

References

[BBFM82] H. Berg, W. Boebert, W. Franta, and T. Moher. *Formal Methods of Program Verification And Specification.* Prentice-Hall, Inc., 1982.

[BF82] A. Barr and E. Feigenbaum, editors. *The Handbook of Artificial Intelligence.* HeurisTech Press, 1982.

[Def85] Department of Defense. *DoD Trusted Computer System Evaluation Criteria.* Technical Report 008-000-00461-7, DoD 5200-28-STD Washington, D.C. Department of Defense, 1985.

[Den82] D. E. Denning. *Cryptography and Data Security.* Addison-Wesley Publishing Company, 1982.

[Fie80] R. J. Fiertag. *A Technique for Proving Specifications are Multilevel Secure.* Technical Report CSL-109, Technical Report, SRI International, 1980.

[FN71] R. Fikes and N. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence,* 2:189–208, 1971.

[Fra83] L. J. Fraim. SCOMP: a solution to the multilevel security problem. *IEEE Computer,* 16(7):26–33, 1983.

[Fri79] P. Friedland. *Knowledge-based experiment design in molecular genetics.* Technical Report 79-771, Computer Science Dept., Stanford University, 1979.

[Lin89] T. A.. Linden. Representing software designs as partially developed plans. *IJCAI Workshop on Automating Software Design,* 1–9, 1989.

[McC88] D. McCullough. *Foundations of ULYSSES: the theory of security.* Technical Report RADC-TR-87-222, Odyssey Research Associates, Inc, July 1988.

[Mil79] J. K. Millen. *Operating System Security Verification.* Technical Report M79-223, The MITRE Corp, Bedford, Mass, 1979.

[Mil87] J. K. Millen. Covert channel capacity. *Proceedings of the 1987 Symposium on Security and Privacy,* 60–66, April 1987.

[Nil80] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Company, 1980.

[NS72] A. Newell and H. Simon. *Human Problem Solving.* Prentice-Hall, Inc., 1972.

[TGC87] C. R. Tsai, V. D. Gligor, and C. S. Chandersekaran. A formal method for the identification of covert storage channels in source code. *Proceedings of the 1987 Symposium on Security and Privacy,* 74–86, 1987.

KUIE LAYOUT — AN INTERVAL-BASED GRAPHICAL CONSTRAINT SYSTEM

Bob Schrag[1]

Honeywell Systems and Research Center
3660 Technology Drive
Minneapolis, MN 55418

schrag@src.honeywell.com

ABSTRACT

The KBSA Framework's User Interface Environment Layout system is a novel application of interval-based constraint technology to the problem of automated user interface design, to provide a highly declarative and flexible tool. In addition to an interval bounds propagator (the Assimilator), Layout includes a special-purpose interval bounds information extractor (the Allocator) to select particular values from allowed intervals and create a presentable picture, and a special-purpose interval bounds reason dependency management system to support incremental change. The interval-based approach gives Layout greater expressive power than a more conventional "flat" quantitative constraint for user interface design, at the cost of higher computational complexity (it is polynomial). We believe there is a body of user interface applications of significant size that will benefit from the flexibility of this technology, using the current generation of hardware workstation technology. This paper describes Layout's expressive capabilities, computational power, and overall software architecture. Layout is implemented in the KUIE Lisp-based graphical object environment; the concepts are transferrable to another environment, in principle.

INTRODUCTION

The Honeywell Systems and Research Center has just completed a four-year contract with RADC to develop integration technology for KBSA. One of our main contributions has been "KUIE"—the KBSA User Interface Environment. KUIE is a Lisp-based object-oriented graphical user interface construction toolkit. It is written in Common Lisp and CLOS and builds its higher-level, declarative capabilities on top of X-Windows, CLX[2], and CLUE[3]. Contributions are also expected to be taken from CLIO[4].

[1]This work was supported by the Rome Air Development Center under U.S. Air Force contract No. F30602-86-C-0074

[2]The Common Lisp interface to X-Windows.

[3]The "Common Lisp User Environment," a Common Lisp interface to the X toolkit, done by Texas Instruments.

[4]Common Lisp Interactive Objects, a toolkit written by Texas Instruments containing basic objects such as menus, sliders, and forms.

KUIE is organized into three levels:

Level 1 — Building Blocks. Facilities for graphical object construction. KUIE Level 1 is a toolkit that extends CLUE graphical objects (contacts). It permits them to be non-rectangular and transparent, and it provides automatic display maintenance.

Level 2 — Layout. Facilites for automatic graphical object sizing and placement.

Level 3 — Gesture Recognition. Facilities for recognizing user-to-graphical-object interactions, so that they can be interpreted in an application program context.

This paper describes KUIE Level 2 — "Layout" — and its distinctive capabilities drawing from the supporting technology of interval-based constraint propagation. Interval-based constraints are more powerful, as well as more computationally expensive, than "flat" numeric constraints.

MOTIVATION, COMPARISON TO OTHER WORK, AND GOALS

KUIE in general has been designed with the intention of alleviating the programmer as much as possible from the low-level "bookkeeping" work that has been common to user interface programming up to this point. KUIE Level 1 — Building Blocks — does a good job of keeping all but the most salient details from the programmer's purview. In Layout, our goal has been to continue in this paradigm of shifting as much burden to the system from the programmer as possible.

Constraint-based graphical layout systems have been experimented with for almost 30 years [1], and there are presently other working constraint-based graphical user interface toolkits, including ThingLab II [2], Coral [3], Constraint Window System [4], Graphical Object Workbench [5], and Constrained Rectangular Tiled Layout [6]. Some of these toolkits also contain provision for "non-layout" (non-geometric) graphical object constraints, such active values relating graphics and other application objects or procedures. With respect to actual "layout" (geometric) constraints, all of these systems are either relatively limited in the scope of their capabilities, dealing only with a specific subset of the total layout problem (such as window tiling), or requiring that the numeric constraints specified be exact. In KUIE, we decided that this restriction of exact numeric values may put too much of a burden on an application programmer, and we wanted to design a more flexible facility. Our overall perception is that users don't like to mess with pixels, and we have wanted to do as much as we can toward taking the "magic number" business out of the programming.

Thus, for example, we wanted the programmer to be able to say, "rectangle-1 is above rectangle-2," without necessarily having to say "how far" above, or how many such specifications may ultimately be made, and have the system do something reasonable. The interval-based constraint satisfaction approach affords the programmer the ability to leave such constraint specifications loose. The programmer may also have notions about the minima and maxima for specific geometric quantities; the interval approach accommodates these specifications, and uses constraint propagation to deduce their widest possible bounds across a complete, specified geometrical layout. The propagation process thus takes the place of potentially tedious and time-consuming, iterative manual placement experimentation. (Presently we are working only with a programmatic specification interface. Conceivably, this could be extended into an interactive constraint specification interface; then the developer's visual sense could be more advantageously exploited, and an even better set of initial constraints for satisfaction obtained. We would like to explore this in the future.) We know of only one other graphical interface design system that uses intervals—an experimental German system that currently uses a very expensive, general constraint satisfaction procedure (simulated annealing) [7].

Our overall architecture plan is broad enough to encompass also the maintaining of constraints during user interaction. The performance of the constraint bounds maintenance system (that enables incremental change) is still probably less than truly interactive—on the order of seconds, rather than of tenths of seconds. Further accelerated-performance workstations and some performance engineering may fix this. The contributions of KUIE Layout in the programmatic regard are nonetheless unique, and powerful.

CAPABILITIES AND ARCHITECTURE

The KUIE Layout system is a programmatic ("software," as opposed to "user") interface to KUIE geometrical object layout definition. The scope of the addressed layout problem includes object size, positioning, and relative placement. Layout is a numerical interval-based constraint specification and satisfaction system that allows upper and lower quantitative bounds to be placed on well-defined object and inter-object geometric properties, *e.g.*, the width of this rectangle should be between 20 and 60 pixels ("`[20 60]`"). (Interval-based constraint propagation is discussed in [8].)

Layout consists of three main existing parts, and one partially completed part:

1. the **Specifier**, including graphical object and constraint definitions;

2. the **Assimilator**, an arithmetic propagation engine that refines all known bounds to their least justified quantitative span, and detects any bounds or constraint inconsistencies; and

3. the **Allocator**, which, upon consistent, completely propagated assimilation, performs remaining allocates interval slop to to fix underconstrained (still interval-valued) bounds at an exact number, and leads to

harmonious object relative placement;

4. the (partially completed) BMS ("bounds maintenance system"), a special purpose, streamlined reason maintenance system to support enhanced interactive operation.

These parts are invoked under program control. Layout is integrated with the rest of KUIE (through mixins to object graphic) and through hooks to the CLUE geometry manager (upon "realization"). A user may choose to use or not use the special capabilites Layout for various parts of his or her problem. (That is, exact numeric bounds may still be used for any geometric quantities, and no objects need necessarily be constrained.)

THE SPECIFIER

For the sake of generality, Layout deals specifically with object rectangle ("bounding box") definitions; other KUIE graphic objects can be defined in these terms. Nested, composite rectangles are dealt with at the level of a unitary constraint system. A Layout rectangle specification has the following information. This serves as input to the Layout system.

```
(setf r1 (make-instance 'rectangle
                        :height [20 30]
                        :width 40
                        :top [0 100]
                        :bottom [0 400]
                        :left 600
                        :right [0 800]))
```

Any initialization field left unspecified is "unconstrained;" that is, it takes an interval value of `[0 :infinity]`.

In addition to the above initialization fields, Layout also includes *handles,* which are 2-dimensional points such as `origin` and `center`, and *parameters,* such as `area` and `aspect-ratio`. Handles and parameters may be defined by the user using special definition forms and then used in constraint specifications.

Some examples of higher-level Layout specifications are shown below. These serve as input to the Layout system.

```
(place r1 :left-of r2 :offset [100 200])
(place r1 :left-of r2 :overlapping [0 25])
(align (top r1) :to (top r3))
(align (y (center r7)) :to (y (center r1)))
(scale (width r7) :to (width r1) :as 2)
(equate (aspect-ratio r3) :with 1)
(scale (area r3) :to (area r2) :as [3/2 5/2])
(tile r7 :vertically :children (list r8 r9 r10))
```

Specifically omitted from Layout "placement" specifications are any notions of negation and disjunction, *i.e.*, "r1 is not above r2," or "r2 is left-of or right-of r1." Including these in general would make the layout problem intractable. This is a compromise: our constraint satisfaction machinery is already polynomial-complexity; adding disjunction would make it exponential. For example, one commonly desired specification capability would be that "r1 is near-to r2 without overlapping." In our system this would have to map into the four-way disjunction: "r1 is either left-of r2 or right-of r2 or above r2 or below r2," with appropriate offset distance specifications in each direction. We have considered the possibility of including such "conditional constraints," and we would like to experiment with structured ways of using them. A utility to help a user understand the computational size of his or her problem and where its complexity comes from would also be useful. We also have a strategy to implement a Layout specification capability for row or column "wrapping" of child objects into an arrayed representation.

In addition to the "standard" types of constraint specifications represented above, we also allow the specification of "arbitrary" arithmetic constraints over geometric quantities, using the operations of addition, subtraction, multiplication, division, square, and square-root. (Indiscriminate use of arbitrary constraint relations, may, however, lead to trouble—see below.)

These Layout capabilities have been selected based on the mix of KUIE applications currently underway or planned at Honeywell; other capabilities may be considered for addition as needs are identified.

THE ASSIMILATOR

The Assimilator is a Waltz-style propagator, as described in [8]. *Geometric* quantities, such as height, area, x-coordinate, or a scale factor, are represented as *nodes* with interval bounds. *Constraints* relate these nodes arithmetically. When constraints are initially asserted or whenever a node which is a member of a constraint is "changed," the constraint is *queued* for refinement processing. In refinement, each constraint node is evaluated arithmetically with respect to the other nodes, and if the evaluation result is "tighter"—with either a higher lower-bound or a lower upper-bound than the node under refinement, then that node is changed to reflect the tighter bounds warranted by the current state of the system. The algorithm terminates when the entire network represented by such constraints becomes quiescient.

Our propagation machinery uses floating-point arithmetic, in order to obtain precision in dealing with division and square-root operations.

Arithmetic interval constraint propagation systems have some cantankerous properties, most of which we have been able to avoid in this relatively restricted problem domain. Under some "pathological" initial conditions, they may not terminate. So far, the "sensible" constraint specifications that we have developed are oriented to intuitive and apparently well-behaved properties of Euclidian geometry, and have not been pathological. Under poor constraint selection ordering, execution time may become exponential. We employ a "stratified" system of separate queues based on different arithmetic constraint types, in order to maintain some control over constraint refinement order. Complexity analysis for interval propagation systems is difficult and depends on the kinds of arithmetic constraint operations employed. So far, we have not done a formal complexity analysis, but our initial experiments show that, for problems tested, execution time grows as the square of the problem size in number of nodes.

The input of assimilation is a set of "unrefined" nodes; its output is a set of nodes with intervals that are consistent with all of the constraints of the system. The Assimilator has some limitations on computational power that also demand the taking of extra care in writing constraint expressions.

In particular, the Assimilator can perform no equality substitution (or "term-rewriting"), either across constraint arithmetic expressions or within single constraints—*even though there clearly are* computational situations where term, or "node," substitutions are necessary to solve for the best bounds on nodes. We made this decision since general term-rewriting mechanisms are intractable for propagation. The danger in this is that a value picked for a node whose assimilated bounds are not the best logically warranted bounds *might not* actually be logically consistent with the rest of the system. This would become a problem in allocation. The way that we have taken out of this computational impovershment is to assert additional constraints that solve equality relationships for you, in effect, performing the required substitutions in them, "by hand." So far, this has been satisfactory: the problem has come up only in constraints employing square and square-root operations, and we have coded it into our "primitives" for rectangle area and point-to-point straight-line distance. A section of our users manual will flag this problem potential, and, at any rate, it only applies to "custom-generated" arbitrary constraints.

THE ALLOCATOR

One remaining challenge to using the interval-based approach has been selecting particular values for each node in the system. The Assimilator only refines interval bounds to the set of what is mutually consistent among all constraints. This, as Davis points out ([8]), *in effect only defines the "Cartesian product"* of the solution space over all nodes; while it is true (given a set of constraints with adequate hand-substituted "rewrites" to make it logically complete) that for any one node a value can be chosen that when taken with the rest of the system is consistent, there is no guarantee that you can do this for two different nodes at the same time. Extracting a particular solution *from* a fully assimilated network still requires additional work. This is what the Allocator is for.

In comparison to the Assimilator, which may be considered as making its refinement decisions at an "object-level," basing them on individual bounds, the

Allocator can be said to work at a "meta-level," basing its decisions on more global information, including the current *delta* for a node between its lower and upper bounds, and the sum of node lower-bounds along a parent edge-to-edge path. Unlike the Assimilator, the Allocator is *non-monotonic* under constraint addition, since assimilated interval bounds and deltas can change under constraint addition.

A very general allocator—one that was always guaranteed to work—would simply loop through all system nodes, choosing an exact value (allocation) for each one, and then reassimilating. Such a process would not necessarily be the most efficient possible, or even generate a very pretty picture. One of our early attempts at an Allocator was a variation on such a process. Our present Allocator uses information about parent-rectangle edge-to-edge child-rectangle paths and path-lengths, in the x- and y-dimensions, to determine an intuitively "centering" allocation that is also relatively efficient, in that it allocates all of the nodes on a path in one step, between assimilations, when possible.

Our approach is to work first on the path with minimum "freespace"—the difference between parent edge-to-edge length and the sum of node interval lower bounds along the path. We give each length node a "fair" allocation, defined as the ratio of its interval delta to the sum of all deltas on the path, times the freespace. By choosing the path with the minimum freespace, we tend to fix its length nodes in a "centered" position, before going on to other, shorter-bounded paths. Choosing these larger-freespace paths first would lead to allocations in which later-allocated rectangles would end up bunched at one end of the picture. Our Allocator includes a minimum-freespace path search algorithm that also caches best paths between assimilations.

These decisions in our Allocator are relatively arbitrary, but they are also straightforward. Any alternative must somehow prioritize nodes for allocation order and also provide formulas, or "allocation directives" that pin down values from intervals.

Allocation performance tends to be slower than assimilation performance, because it is a multi-pass assimilation process. For the example problem listed below, there is about a factor of two difference.

THE BMS — BOUNDS MAINTENANCE SYSTEM

The BMS is a special-purpose, streamlined reason maintenance system to support fast incremental change and enhanced interactive operation. Interval bounds systems respond well to addition of new constraints—propagation is monotonic through assimilation—but without special provision they include no support for constraint deletion or retraction without complete network re-assimilation. Our Allocator, being non-monotonic, requires a BMS to support even addition incrementally. The BMS works by recording constraint-and-node dependencies for all node refinements and allocation decisions. It is streamlined in that it simply discards values that go "out," rather than ever letting them come back "in," through backtracking. The BMS concept is also essential in the (possible) *implementation of conditional constraints*, since it will supply the machinery for undoing propagation, and also as the basis for more robust constraint violation exception handling.

Our initial experiments with the BMS concept showed that it might be a feasible approach, with reasoning as follows. It is, typically, much less expensive to add a new constraint to an already assimilated system and then to reassimilate than it is to reassimilate the whole system over again from scratch. By analogy, the "rollback" of a deleted constraint should also be, typically, of a much smaller order than the reassimilation process too. Taken together (in sequence), these two processes—deletion/rollback plus incremental reassimilation—should still be significantly less expensive than wholesale reassimilation. Now this has been our experience in practice.

We have prototyped a BMS, and we have successfully tested it through the assimilator, performing retraction and reassimilation. Our BMS prototype also works for retraction through the Allocator, but we have no remaining funds to complete allocation reassimilation. One basic question in BMS development was about its performance. The overhead of dependency recording for the BMS was smaller than originally anticipated—about 15%. While it is impossible to make general claims about incremental change performance, because of the idiosyncrasies of different problem specifications, we have performed a combination of experimentation and analysis regarding different retraction and reassimilation instructions for one interesting problem—the example listed below. We have found that, through assimilation, rectangle deletion and reassimilation is proportional to the relative number of constraints included: r1, which includes about half of the problem's constraints, takes about half of the problem's assimilation time to retract and reassimilate; r7, which has about one-tenth of the constraints, takes about one-tenth of the assimilation time. These results, while somewhat expected, are encouraging.

Our experiments with allocation retraction performance seem to demonstrate a more pointed dependence on individual problem idiosyncrasies: since allocation is very much a sequential process (pick minimum-freespace path, allocate, and assimilate—in a loop), the order of allocation is important to subsequent dependencies and retraction performance, and those rectangles which are allocated earliest and most "centered" tend to be the most critical to growing retraction time. This is not a completely damaging result necessarily, because there are circumstances under which allocation can be controlled or isolated to specific display regions, as in the inside of a composite object since allocation takes place through successive composite levels. Top-level changes have wholesale effects; local changes have local effects. There will certainly be cases where it is smarter simply to reallocate from the pre-allocated assimilated state than to retract allocations, reassimilate, and reallocate, or to completely reassimilate and reallocate from the originally specified state. Whether generally sensible cutoffs for these decisions can be determined is not yet clear.

EXAMPLE

Here is an example KUIE Layout specification, which serves as input to the Layout system.

```
  (let ((wd [40 50])
        (ht [20 30])
        (parent (make-rectangle :width 800 :height 600)))
    (make-rectangle 'r1 :width wd :height ht :parent parent)
    (make-rectangle 'r2 :width wd :height ht :parent parent)
    (make-rectangle 'r3 :width wd :height ht :parent parent)
    (make-rectangle 'r4 :height ht :parent parent)
    (make-rectangle 'r5 :parent parent)
    (make-rectangle 'r6 :parent parent)
    (make-rectangle 'r7 :parent parent)
    (place r1 :above r2)
    (align (x (center r1)) :to (x (center r2)))
    (place r3 :left-of r1 :overlapping t)
    (place r4 :above r1)
    (align (center r4) :to (left r1))
    (scale (width r4) :to wd :by 3)
    (place r5 :right-of r1)
    (align (y (center r5)) :to (y (center r1)))
    (scale (width r5) :to (width r1) :by [1 2])
    (scale (height r5) :to (height r1) :by [2 3])
    (place r6 :right-of r5)
    (align (y (center r6)) :to (y (center r5)))
    (equate (aspect-ratio r6) :with [1 2])
    (scale (area r6) :to (area r5) :by [2 3])
    (scale (left r7) :to (left r5) :by [3 5]))
```

It creates seven child rectangles. The first three share common width nodes and the first four share common height nodes, from the lexical variables wd and ht. Three heights, four widths, and all coordinates are unconstrained. Most of the specifications are straightforward; in the second scaling constraint, (width r5) is scaled to (be, in this case, bigger than) (width r1) by a factor of from [1 2]. The "equating" specification identifies r6's aspect ratio with an interval.

Immediately after specification, the seven rectangles have nodes with the following bounds. This is the output from the Specifier. ("INF" is short for "INFINITY.")

```
(("R1" :WIDTH [40 50] :LEFT [0 :INF] :RIGHT [0 :INF]
       :HEIGHT [20 30] :TOP [0 :INF] :BOTTOM [0 :INF]
       :HANDLES ((CENTER :X [0 :INF] :Y [0 :INF])))
 ("R2" :WIDTH [40 50] :LEFT [0 :INF] :RIGHT [0 :INF]
       :HEIGHT [20 30] :TOP [0 :INF] :BOTTOM [0 :INF]
       :HANDLES ((CENTER :X [0 :INF] :Y [0 :INF])))
 ("R3" :WIDTH [40 50] :LEFT [0 :INF] :RIGHT [0 :INF]
       :HEIGHT [20 30] :TOP [0 :INF] :BOTTOM [0 :INF])
 ("R4" :WIDTH [1 :INF] :LEFT [0 :INF] :RIGHT [0 :INF]
       :HEIGHT [20 30] :TOP [0 :INF] :BOTTOM [0 :INF]
       :HANDLES ((CENTER :X [0 :INF] :Y [0 :INF])))
 ("R5" :WIDTH [1 :INF] :LEFT [0 :INF] :RIGHT [0 :INF]
       :HEIGHT [1 :INF] :TOP [0 :INF] :BOTTOM [0 :INF]
       :HANDLES ((CENTER :X [0 :INF] :Y [0 :INF]))
       :PARAMETERS ((ASPECT-RATIO [:-INF :INF])
                    (AREA [:-INF :INF])))
 ("R6" :WIDTH [1 :INF] :LEFT [0 :INF] :RIGHT [0 :INF]
       :HEIGHT [1 :INF] :TOP [0 :INF] :BOTTOM [0 :INF]
       :HANDLES ((CENTER :X [0 :INF] :Y [0 :INF]))
       :PARAMETERS ((AREA [:-INF :INF])
                    (ASPECT-RATIO [:-INF :INF])))
 ("R7" :WIDTH [1 :INF] :LEFT [0 :INF] :RIGHT [0 :INF]
       :HEIGHT [1 :INF] :TOP [0 :INF] :BOTTOM [0 :INF]))
```

Except for the width and height nodes we set explicitly, all nodes are unconstrained, except for system-defined limits.

After assimilation, the rectangles' nodes have the following bounds. This is the output from the Assimilator.

```
(("R1" :WIDTH [40 50] :LEFT [60 226] :RIGHT [100 266]
       :HEIGHT [20 30] :TOP [20 560] :BOTTOM [40 580]
       :HANDLES ((CENTER :X [80 246] :Y [30 570])))
 ("R2" :WIDTH [40 50] :LEFT [55 226] :RIGHT [100 271]
       :HEIGHT [20 30] :TOP [40 580] :BOTTOM [60 600]
       :HANDLES ((CENTER :X [80 246] :Y [50 590])))
 ("R3" :WIDTH [40 50] :LEFT [10 226] :RIGHT [60 266]
       :HEIGHT [20 30] :TOP [0 580] :BOTTOM [20 600])
 ("R4" :WIDTH [120 150] :LEFT [0 166] :RIGHT [120 301]
       :HEIGHT [20 30] :TOP [0 540] :BOTTOM [20 560]
       :HANDLES ((CENTER :X [60 226] :Y [10 550])))
 ("R5" :WIDTH [40 100] :LEFT [100 266] :RIGHT [140 366]
       :HEIGHT [40 90] :TOP [0 550] :BOTTOM [50 600]
       :HANDLES ((CENTER :X [120 316] :Y [30 570]))
       :PARAMETERS ((ASPECT-RATIO [0 2]) (AREA [1600 9000])))
 ("R6" :WIDTH [40 329] :LEFT [140 760] :RIGHT [180 800]
       :HEIGHT [40 164] :TOP [0 550] :BOTTOM [50 600]
       :HANDLES ((CENTER :X [160 780] :Y [30 570]))
       :PARAMETERS ((AREA [3200 27000]) (ASPECT-RATIO [1 2])))
 ("R7" :WIDTH [1 500] :LEFT [300 799] :RIGHT [301 800]
       :HEIGHT [1 600] :TOP [0 599] :BOTTOM [1 600]))
```

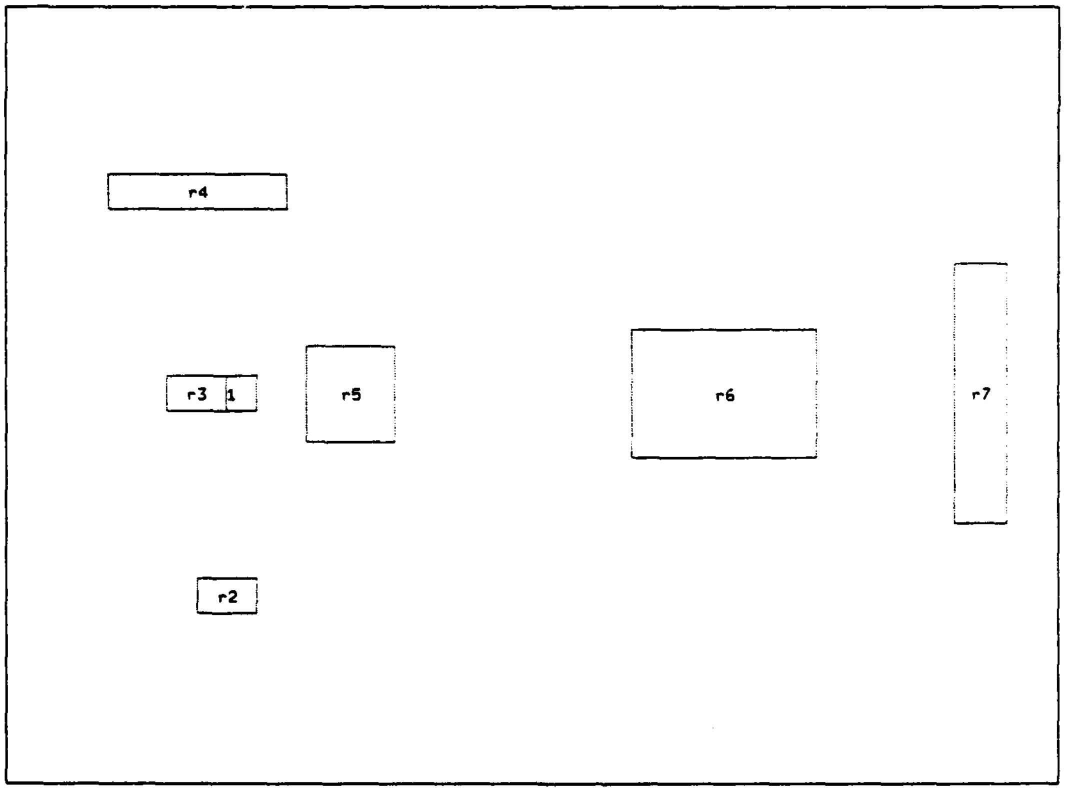
These are the widest bounds that are consistent with all of the input specifications. Note that all of the child rectangles' coordinates fit within the dimensions of the parent rectangle.

Now the bounds must be allocated. This is the only way to get a firm, meaningful picture to display. After allocation, the nodes have the following values.

This is the output from the Allocator.

```
(("R1" :WIDTH 44 :LEFT 144 :RIGHT 189
       :HEIGHT 28 :TOP 285 :BOTTOM 314
       :HANDLES ((CENTER :X 167 :Y 300)))
 ("R2" :WIDTH 44 :LEFT 144 :RIGHT 189
       :HEIGHT 28 :TOP 442 :BOTTOM 471
       :HANDLES ((CENTER :X 167 :Y 457)))
 ("R3" :WIDTH 44 :LEFT 121 :RIGHT 166
       :HEIGHT 28 :TOP 285 :BOTTOM 314)
 ("R4" :WIDTH 134 :LEFT 77 :RIGHT 211
       :HEIGHT 28 :TOP 128 :BOTTOM 157
       :HANDLES ((CENTER :X 144 :Y 143)))
 ("R5" :WIDTH 79 :LEFT 215 :RIGHT 295
       :HEIGHT 82 :TOP 258 :BOTTOM 341
       :HANDLES ((CENTER :X 255 :Y 300))
       :PARAMETERS ((ASPECT-RATIO 1) (AREA 6528)))
 ("R6" :WIDTH 181 :LEFT 470 :RIGHT 652
       :HEIGHT 104 :TOP 247 :BOTTOM 352
       :HANDLES ((CENTER :X 562 :Y 300))
       :PARAMETERS ((AREA 18953) (ASPECT-RATIO 2)))
 ("R7" :WIDTH 61 :LEFT 677 :RIGHT 739
       :HEIGHT 200 :TOP 199 :BOTTOM 400))
```

The allocated configuration is shown in this figure. This example makes rather



liberal use of interval-valued and unconstrained nodes, with only one scale factor pinned down to an exact value before assimilating. In general, the more allocated values there are in a system specification, the faster it will go; this efficiency differential also applies to dependency management and incremental change.

Quantitative performance information for the example is as follows. Execution times are for a Sun-4 SPARC-station 1+ running Allegro Common Lisp.

- There are 111 constraints asserted.
- There are 144 interval nodes.
- Specification runs in about 3 seconds.
- Assimilation takes about 4.5 seconds.
- Allocation takes about 9 seconds.

This total problem execution time of about 16.5 seconds—higher than our average rate reported below, but for a more-than-average complicated problem.

STATUS AND FUTURE GOALS

The Specifier, Assimilator, Allocator, and BMS are all existing and working in initial implementations. Execution times for the sum of the three constructive operations appear to be on the order of 1 second per object, for fairly large (50-rectangle) constraint systems, running on the Sun-4 Sparc-station. We are using KUIE at Honeywell on a variety of in-house projects and contracts, our further work is presently limited to a kind of maintenance and support role, and we are actively seeking additional funding sources and opportunities. When we find these, our near-term goals for Layout are BMS completion through the Allocator, general performance enhancement (the system is not yet highly "tuned"), and full integration with the rest of KUIE. Long-term goals include investigation of interactive and visual constraint representations, constrained object "gesture interpretation" (KUIE Level 3), and robust exception handling/conditional constraints.

SUMMARY

We have utilized interval-based contraint technology to produce a user interface design tool that is declarative and flexible—KUIE Layout. This tool will significantly alleviate interface designers' burdens from the mundane, non-declarative pixel-and-number-level programming details that characterize much user interface programming, and it will make the resultant interface code much more flexible under user interactive operations and more adaptable under user interface system evolution.

We believe that this user interface work has answered some previously unanswered technical questions. As we were beginning this work, we were aware of three outstanding questions about interval-based constraint technology that had been posed by Davis [8]:

1. Can useful information be extracted from an interval label system?
2. Can the technology be usefully applied in the spatial domain?
3. Is there a workable approach to incremental change (deletion)?

We believe that KUIE Layout represents a sucessful demonstration to answer each of these questions in the affirmative: the Allocator neatly extracts useful information from the node intervals; the graphical layout problem which we have worked on is clearly spatial; and our BMS is apparently workable, but we must wait for a more substantial of it to assess its full utility.

REFERENCES

[1] Sutherland, I., "Sketchpad: A Man-machine Graphical Communication System," *Proceedings of the Spring Joint Computer Conference* (IFIPS), 1963, 329–345.

[2] Maloney, J.H., Borning, A., and Freeman-Benson, B.N., "Constraint Technology for User-Interface Construction in ThingLab II," *Proceedings of the ACM Conference on Object-oriented Programming Systems, Languages, and Applications* (OOPSLA), 1989, 381–388.

[3] Szekely, P.A. and Myers, B.A., "A User Interface Toolkit Based on Graphical Objects and Constraints," *Proceedings of the ACM Conference on Object-oriented Programming Systems, Languages, and Applications* (OOPSLA), 1988, 36–45.

[4] Epstein, D. and LeLonde, W.R., "A Smalltalk Window System Based on Constraints," *Proceedings of the ACM Conference on Object-oriented Programming Systems, Languages, and Applications* (OOPSLA), 1988, 83–94.

[5] Barth, P.S., "An Object-oriented Approach to Graphical Interfaces," *ACM Transaction on Graphics,* 5:2, April 1986, 142–172.

[6] Cohen, E.S., Smith, E.T., and Iverson, L.A., "Constraint-based Tiled Windows," *IEEE Computer Graphics and Applications,* May 1986, 35–45.

[7] Bolz, D., personal communication, GMD.

[8] Davis, E., "Constraint Propagation Through Interval Labels," *Artificial Intelligence* 32:3, July 1987, 281–331.

[9] Dean, T.L. and McDermott, D.V., "Temporal Data Base Management," *Artificial Intelligence* 32:1, April 1987, 1–55.

Multi-Agent Rule-Based Software Development Environments

Naser S. Barghouti[*] and Gail E. Kaiser[†]

Department of Computer Science, Columbia University
New York, NY 10027
naser@cs.columbia.edu, (212) 854-8182
kaiser@cs.columbia.edu, (212) 854-3856

Abstract

Researchers have explored the possibility of applying production systems to the domain of software development. One resulting approach is rule-based development environments (RBDEs), which provide expert assistance to developers working on large-scale projects. RBDEs model the development process in terms of rules, and then "enact" this model by automatically firing rules at the appropriate time to carry out chores that the developer would have otherwise had to do manually. In order to realistically model the domain of software development, RBDEs must support *cooperation* among multiple developers, each of whom selects commands, causing the firing of multiple rules (either directly or via chaining) that concurrently access shared information. One of the problems resulting from executing rule chains concurrently is how to detect and resolve conflicts that occur between the chains. This paper presents the MARVEL rule-based development environment, explores the concurrency control problem in a multi-agent model of MARVEL, and suggests an approach to solving it.

[*]Barghouti is supported in part by the Center for Telecommunications Research.

[†]Kaiser is supported by National Science Foundation grants CDA-8920080, CCR-8858029 and CCR-8802741, by grants from AT&T, BNR, Citicorp, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

Introduction and Motivation

Production systems have been applied to domains such as medical diagnosis, mathematical discovery, and hardware configuration [BF81]. Recently, researchers have explored the application of production systems to the domain of managing software development efforts, i.e., monitoring the development process and/or enforcing a specific model of development [RB89]. This is distinct from applying production systems techniques to automatic programming and algorithm construction, for automatically generating programs and/or algorithms from specifications.

Techniques from production systems such as rule-based modeling and chaining are an enabling technology for reasoning about and automating parts of the software development process. Rule-based development environments (RBDEs) assist in software development by *enacting* the development process modeled in terms of rules. Enacting a process model involves reasoning about the rules and automatically firing rules as appropriate in order to perform development and data management chores that would otherwise be done manually. Although such integration of rule-based process modeling with database capabilities has recently gained popularity [Per89], existing RBDEs do not scale up to "real" projects.

Personnel in real projects typically work concurrently on separate components, but often share knowledge about project status and overall progress as well as data such as software libraries. This data is highly structured and stored in a common project database. It is necessary to maintain the consistency of the project database without obstructing cooperation and sharing of information among developers. Existing RBDEs do not scale up, since most are either single-user environments that do not allow concurrency or multi-user environments that guarantee consistency by isolating the developers, thus preventing collaborative work.

We investigate the scaling up of MARVEL, an RBDE developed at Columbia University, to support cooperation among multiple developers. Our goal is to build an environment that supports collaborative work in the domain of software development. Our approach is to extend MARVEL with a conflict resolution mechanism that maintains data consistency while permitting cooperative concurrent access to the data. The resulting multi-agent MARVEL model will provide more powerful assistance in the management of software development projects. Our mechanism consists of three components: conflict detection, conflict resolution and a consistency specification rule base. In this paper we concentrate on conflict detection.

The MARVEL Model

Several well-known software development environments have applied rule-based technology to the domain of software development. The CommonLisp Framework (CLF) [CLF88] supports rule-based modeling of the software development process through consistency and automation rules [Coh86]. Refine [SKW85], an automatic transformation system for the purpose of program synthesis, also provides a limited form of automation in the style of CLF. Darwin [MR88] restricts what programmers can do by treating rules as constraints and automating checking and enforcing of these constraints. Grapple [HL88] uses rules to do planning and plan recognition in order to monitor a user-specified process model. The Workshop system [Cle88] distinguishes between six kinds of rules that are used to coordinate the activities of teams of programmers as well as to automate some of their chores. Our own system, MARVEL, enacts the development process of a particular project by automating part of the development process. The architecture and design of MARVEL are described in [KFP88] and [KBFS88]. The latest version of the system, MARVEL 2.5, has been fully implemented and documented; our experience developing and using it is described in [KBS90]. We briefly describe MARVEL's enaction model here.

MARVEL enacts the development process of a project by automatically firing user-specified rules that encapsulate development activities. These activities manipulate the different components of the project (e.g., source code, test suites, documentation, etc.), which are stored in a common project database. Each component is abstracted as an object with a set of attributes reflecting the object's status. The user-level commands correspond to rules. Each rule has a *condition* that must be satisfied before the second part, the development activity the rule encapsulates, is executed. The condition is essentially a query (i.e., read operations) on the status of objects in the project database. The activity is modeled as a "black box" whose inputs and outputs are known, but in order to determine which of several possible outputs it will produce, the black box must be executed. The third part is a set of mutually exclusive *effects*, each of which changes (i.e., writes) the values of object attributes in the database. Which effect to assert depends on the results of the rule's activity.

If the effects of a rule change the objects in the database in such a way that the conditions of other rules become satisfied, those rules are fired automatically. This behavior is termed *forward chaining* and is implemented in OPS5 production systems [For81]. Alternatively, if a condition of a rule is not satisfied, *backward chaining* is performed to attempt to satisfy it. The backward chaining model is implemented in theorem provers, constraint systems, and some production systems [Sho76].

MARVEL is currently a single-user environment since it allows only one developer to request only one command at any one time. The developer must wait until all chaining resulting from her command is finished. In addition, rules are fired sequentially, thus guaranteeing that only one database activity will be in progress at any one time.

Multi-Agent MARVEL

In order to support large-scale projects, MARVEL must be enhanced to allow multiple developers to cooperate on a project. These developers will share a common database that contains all the objects of the project, and they start concurrent sessions in order to complete their specific assignments. During their sessions, the developers will concurrently request operations that access objects in the shared project database. There will be a need to synchronize the concurrent activities because they might introduce *conflicts* that violate the consistency of the objects they access (e.g., if they concurrently change either the same attribute or dependent attributes of the same object or of related objects).

Synchronization in this case does not involve only the human developers but also rules that automatically perform operations on their behalf, since those rules also access the shared database. The term *agent* is used in this paper to refer to both human developers and rules that are fired automatically by MARVEL on their behalf. What is lacking in the existing MARVEL system is the ability to synchronize concurrent accesses by multiple agents to shared data while still providing an environment that supports cooperation among the agents (called *cooperative environment* hereafter).

The problem of concurrent rule executions has been addressed previously, for the purpose of speeding up the execution of production systems through the use of parallelism. However, both Gupta [Gup86] and Miranker [Mir86] have concluded that the speed up that can be expected from parallelism is quite limited in the context of OPS5 production systems that are implemented using the Rete algorithm [For82]. We address a different problem that sounds superficially similar. In our model, parallelism is *intrinsic* in the application domain and not a technique used for speeding up the performance of the expert system.

The concurrency control problem has also been addressed in traditional database management domains such as banking. In these domains, there is a lack of knowledge about the application-specific semantics of database operations, and a need to design general mechanisms that cut across many potential applications. Thus, the best a database management system (DBMS) can do is to abstract all operations on a database to read and write operations. All computations are then programmed into *transactions* that consist of a sequence of read and write operations. Each transaction, if executed atomically (i.e., either all of its operations are performed in order or none are), transforms the database from one consistent state to another. When multiple transactions run concurrently, the DBMS can guarantee that the database is transformed to a consistent state with respect to reads and writes by allowing only executions of the concurrent transactions that are equivalent to a serial execution. This correctness criterion is referred to as *serializability* [BHG87].

Existing multi-user RBDEs use serializability to synchronize concurrent agents by isolating these agents, so that one agent can see only the final results of another agent's work after it has been completed, and cannot share data and/or knowledge with other concurrent agents. Isolation guarantees serializability, and thus strict maintenance of consistency, since it makes each agent's work appear as an atomic transaction. Unfortunately, isolation between concurrent agents unnecessarily prevents cooperation.

Our conjecture is that MARVEL can provide a cooperative environment if it is given knowledge about what it means for the data of a specific project to be in a consistent state, and about the semantics of operations performed by agents on the database. Because of

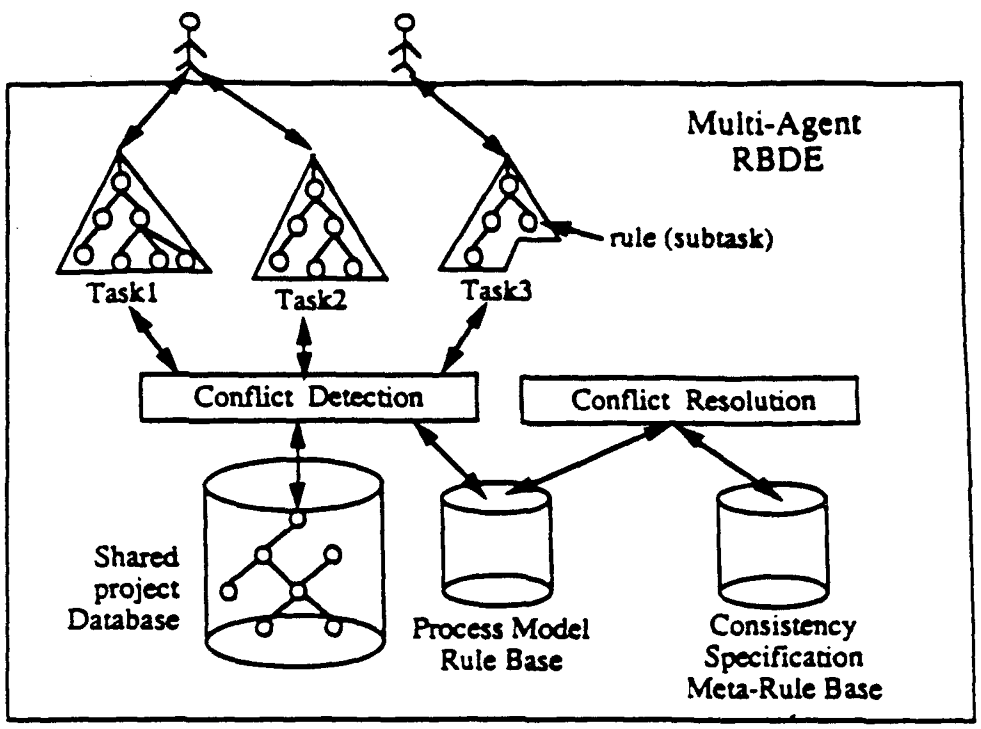


Figure 1: The Multi-Agent Problem in MARVEL

the different meanings of consistency for different projects, it is necessary for the project administrator to provide MARVEL with the consistency specification of the project rather than building-in a specific meaning of consistency that might not apply to all projects.

Synchronizing Multiple Agents in MARVEL

The behavior of cooperating developers in MARVEL can be modeled as concurrent agents, each of which fires multiple rules that may execute concurrently to perform operations on the shared project database. Based on this, we divide the synchronization problem into three subproblems:

- **Conflict detection**: deciding whether or not two or more rules can be fired within concurrent chaining cycles without potentially introducing read/write inconsistencies in the project database. In other words, MARVEL must make sure that the conditions and effects of concurrent rules do not access objects in a conflicting way.

- **Consistency specification**: specifying what kind of consistency has to be maintained in the database of the particular project, and what kind of concurrency is allowable between agents.
- **Conflict resolution**: once a potential conflict has been detected, the RBDE has to decide how to resolve it. The conflict resolution strategy depends on the consistency specification in that if the conflict does violate the specified consistency, then it has to be resolved, but if the detected potential conflict does not violate the consistency specification, it can be ignored.

The different components of the multi-agent problem and the interactions among them are depicted in figure 1. This framework is an extension of the semantics-based consistency suggested by Garcia-Molina [GM83] and Lynch [Lyn83] for traditional database management transactions. In semantics-based consistency, the behavior of each transaction must be known before hand, and it must be possible to statically define the allowed interleavings between groups of transactions. In our software development domain, it is not possible to have such knowledge *a priori* because software developers figure out what activities they will perform as they go along, and even the results of rule chains cannot be known in advance when there are multiple, mutually exclusive, effects. Thus, our tasks are interactive, incrementally constructed, and non-deterministic. The result is that the synchronization problem is more complex than for database management domains.

In the rest of the paper, we present the conflict detection problem and sketch an approach to solve it. We then outline our knowledge-based consistency specification model and the conflict resolution mechanism based on it.

Detecting Conflicts Between Concurrent Agents

We present the conflict detection problem by means of an example. Say that Bob and Mary want to test module ModA by running a test suite, S. ModA consists of two procedures, p1 and p2. The objects (the components of the module and the test suite) are stored in a shared database. Since all user commands (e.g., `test`, `edit`, `compile`, `format`, etc.) are implemented in terms of rules, and since RBDEs provide forward and backward chaining among these rules, each command can cause the firing of a chain of rules, each of which may access objects in the database.

When Bob and Mary request commands concurrently, Bob's command might trigger a chain of rules, one or more of which might conflict with one or more of the rules in the in-progress chain that Mary's command has triggered[1]. Say that Bob requested a command

[1]Alternatively, the conflict might have occurred even in a single-user context if, for example, Bob had requested two commands concurrently (e.g., in two different windows).

```
rule test[?mod: MODULE; ?test: TEST]:
  # We can test a module only after all of its component procedures
  # have been tested.  If the module passes the test, set its
  # test-status attribute to 'tested'; otherwise, to 'failed'.
  (forall PROCEDURE ?p such that (contains ?mod ?p):
              (?p.test-status = tested))
     {TEST test-mod ?test ?mod}
  (?mod.test-status = tested);
  (?mod.test-status = failed);

rule test [?proc: PROCEDURE; ?test: TEST]:
  # To test a procedure, make sure that it is available.
  (?proc.availability = available)
     {TEST test-proc ?test ?proc}
  (?proc.test-status = tested);
  (?proc.test-status = failed);

rule edit [?proc: PROCEDURE; ?user: USER]:
  (and (?proc.availability = reserved)
        (?proc.reserver = ?user))
     {EDITOR edit ?proc}
  (?proc.status = changed);

rule reserve [?obj: OBJECT; ?user: USER]:
  (?obj.availability = available)
     {RCS reserve ?obj ?user}
  (and (?obj.availability = reserved) (?obj.reserver = ?user));
```

Figure 2: Example of MARVEL Rules

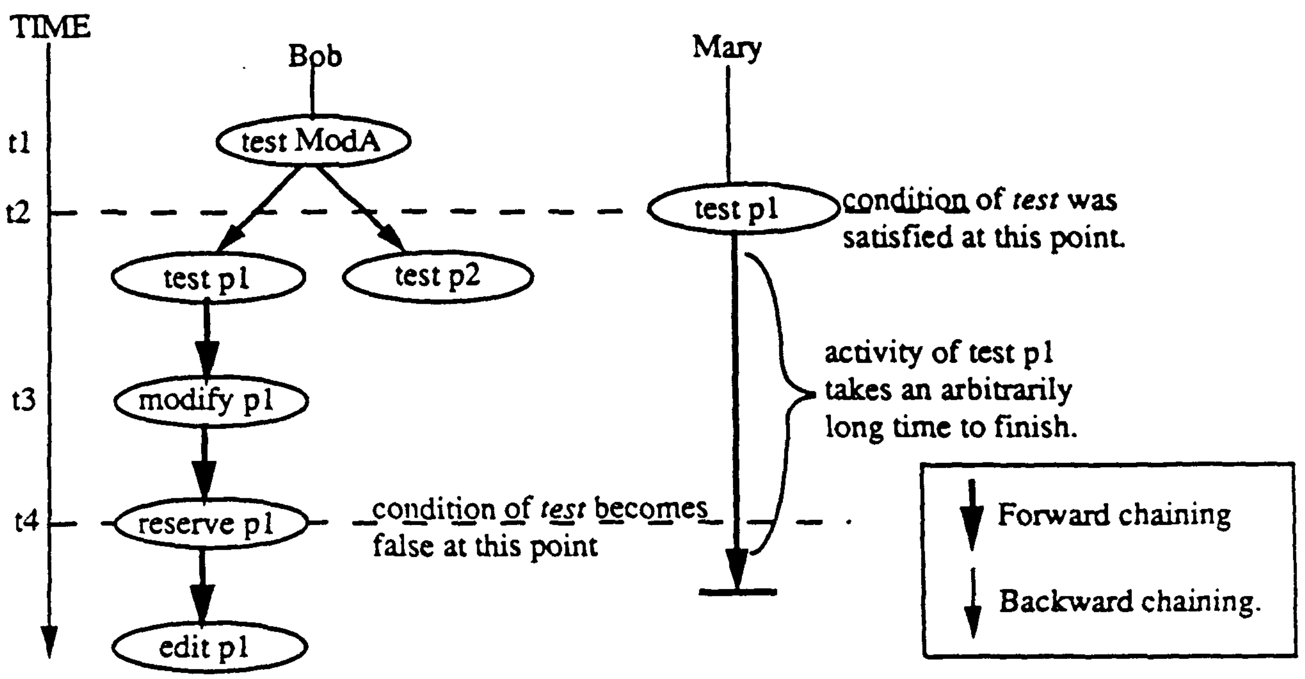

TIME
t1
t2
t3
t4
Bob
Mary
test ModA
test p1
test p2
modify p1
reserve p1
edit p1
test p1
condition of *test* was satisfied at this point.
activity of test p1 takes an arbitrarily long time to finish.
condition of *test* becomes false at this point
Forward chaining
Backward chaining.

Figure 3: Example of the conflict detection problem

`test modA` (corresponding to the `test` rule shown in figure 2) at time `t1`, which triggered the forward chaining cycle shown in figure 3. Mary requests a command `test p1` (corresponding to the second test rule in figure 2) at time `t2`. The condition of the rule is satisfied at that time, causing the activity of the rule to be invoked. In the meanwhile Bob discovers that p1 has a bug so he starts modifying it by initiating the command `modify p1` at time `t3`, which triggers a backward chain to `reserve p1` before calling the editor on p1. From the sequence of activities depicted in figure 3, it should be clear that a conflict results because the effects of the `reserve` rule that Bob's command chains to will negate the condition of Mary's `test` rule, which is already in-progress. This causes Mary's test to be invalid since the procedure she is testing has already been modified.

Tasks as Conflict Detection Units

To detect conflicts similar to the one presented above, each chaining cycle that is triggered in response to a single developer's command is encapsulated in a transaction-like unit called a *task*. Tasks are made up of the set of rules that are fired by the RBDE automatically in response to a developer's command either directly (i.e., the rule corresponding to the command) or indirectly (i.e., through chaining) during a session. Each rule in the task is a *subtask* of the task representing the whole chain. Tasks are both interactive and incremental since their behavior is made up as rules are fired and their effects are asserted (e.g., in the example in figure 3, the RBDE had no way of guessing that Bob will discover a bug that will initiate the chain).

A task is akin to a *nested transaction*, which is a composition of a set of subtransactions, each of which can be a nested transaction [Mos85]. To other transactions, the top-level nested transaction appears as a normal atomic transaction. Internally, however, subtransactions are run concurrently and their actions are synchronized by an internal mechanism. The more important point is that a subtransaction can fail and be restarted or replaced by another subtransaction without causing the whole nested transaction to fail or restart. Similarly, a subtask (rule) within a task (rule chain) can fail and be replaced by another subtask without causing the whole chain to be invalid. This is useful for automation because the RBDE might decide to execute a rule, thinking that it is ready to be executed, only to find out that its condition is not satisfiable. In this case, the rule has to be abandoned and other rules explored. Certainly, we do not want the whole automation cycle to be invalidated because of this failure.

A conflict between concurrent tasks occurs when the firing of a rule (subtask) within one of the tasks violates the serializable execution of an in-progress task. An execution is said to be serializable if an equivalent serial execution can be found. The conflict detection mechanism detects a violation of serializability by using a protocol called *nested incremental locking* (NIL) based on the standard two-phase locking (2PL) protocol used in most traditional DBMSs [EGLT76].

In the example of figure 3, when Bob requests the `test ModA` command, the RBDE starts up a task T_{Bob} and fires the corresponding `test` rule as a subtask. Before the activity of the rule is invoked, T_{Bob} acquires a read lock on the test suite S and on all the procedures contained in ModA (i.e., p1 and p2) and a write lock on the object representing ModA. Similarly, when Mary requests the command `test p1`, T_{Mary} will acquire a write lock on p1. Now when Bob's task fires `modify p1`, it tries to acquire a write lock on p1, but discovers that p1 has already been locked by T_{Bob} and T_{Mary} in an incompatible mode, thus causing both an intra-task conflict (because T_{Bob} has already acquired a read lock on p1 while Bob was testing it) and an inter-task conflict.

Intra-task conflicts are not considered serious and do not cause problems since they can be resolved by either aborting one of the conflicting rules within the same task or ignoring the conflict (e.g., in the example, it does not matter that Bob had locked p1 before). Inter-task conflicts, however, are considered serious, and once detected, they must be resolved by consulting the conflict resolution mechanism, which uses the consistency specification of the particular project to decide on how to resolve the conflict (e.g., in the example, it might be OK for both Bob and Mary to edit and test p1 at the same time because Mary might be interested in testing a feature of p1 that Bob's modification does not change; her test is still valid in this case). We now briefly outline the conflict resolution mechanism.

Resolving Conflicts

Unlike in traditional database management, detecting conflicts does not automatically mean that one of the tasks involved in the conflict has to be aborted, as demonstrated by the previous example. The conflict detection mechanism presented above detects all potential conflicts and leaves it up to the conflict resolution module to consult the specific consistency requirements of the project and decide if the detected conflict actually violates any of them.

In RBDEs, some inconsistency can be tolerated as a price for allowing more concurrency, and thus cooperation, between concurrent tasks. The level of tolerable inconsistency depends on the operations in a particular application. The problem is finding a framework for specifying which interleavings between concurrent tasks are allowable in a particular environment. The framework should specify the *granularity* at which different database operations (performed to either evaluate the condition of a rule or assert its effects) can be interleaved. This specification framework can then be used by the conflict resolution algorithm to provide maximum concurrency while maintaining consistency.

Our approach is to specify the consistency constraints of a project in terms of a number of *meta-rules*, each with a condition and an action. The condition defines a conflict situation that involves specific rules and on-going tasks. The action describes how the conflict is resolved through a repair mechanism. If there is no matching meta-rule, then the potential conflict is permitted, e.g., when the optional automation of RBDEs, but not consistency, is affected. Another approach is to disallow the conflict and abort the transaction containing the rule that caused the conflict is a matching meta-rule is not found.

The conflict resolution protocol consults the meta-rule base whenever a potential conflict is detected, to determine whether a particular interleaving is permitted. If not, and there is a repair specified (e.g., as a sequence of operations on the database), the repair is attempted. If there is no repair, or if the repair is not successful (the repair may fail if its operations are disallowed due to other conflicts), then the transaction containing the conflicting rule must be aborted. The repair mechanism provides a means for avoiding rollback of transactions that might otherwise be aborted, in the sense that all updates are forgotten and the database restored to the state it was in before the transaction began; instead, a meta-rule might specify a compensation function that undoes the semantic effects of the rule chain or carries out some other activity. This is important for our RBDE domain, where much human effort might go into database operations, and automatically throwing away this work due to a conflict would be unacceptable.

Conclusions

We considered the problem of applying techniques from rule-based production systems to software development environments. Rule-based specification of the development process enables software development environment to automate parts of the software development process. Supporting the development of large software project introduces the problem of synchronizing the concurrent activities of multiple developers and the rule chains fired on their behalf, collectively termed multiple agents. We formulated an approach for detecting conflicts between concurrent agents. Our approach is based on grouping chains of rules in transaction-like units called *tasks* and defining the NIL protocol for detecting potential conflicts between concurrent tasks. Finally, we briefly explained how a conflict resolution mechanism uses a project-specific consistency specification to resolve potential conflicts detected by the NIL protocol.

Acknowledgments

Michael Sokolsky collaborated with us on designing and implementing MARVEL and its object management system. George Heineman has worked with us in designing consistency rules for MARVEL . Israel Ben-Shaul has implemented the rule selection mechanism in MARVEL and is now the research staff managing the project. MARVEL 2.6 is now available for licensing to educational institutions and industrial sponsors; please contact Israel Ben-Shaul (israel@cs.columbia.edu) for more information.

References

[BF81] A. Barr and E. Feigenbaum. *The Handbook of Artificial Intelligence, Volume 1.* William Kaufmann, Inc., Los Altos CA, 1981.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading MA, 1987.

[Cle88] G. M. Clemm. The workshop system – a practical knowledge-based software environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 55–64, Boston, November 1988. ACM Press. Special issue of *SIGPLAN Notices*, 24(2), February 1989 and of *Software Engineering Notes*, 13(5), November 1988.

[CLF88] University of Southern California, Information Sciences Institute, Marina del Rey CA. *CLF Manual*, January 1988.

[Coh86] D. Cohen. Automatic compilation of logical specifications into efficient programs. In *Proceedings of the 5th National Conference on Artificial Intelligence*, volume Science, pages 20–25, Philadelphia PA, August 1986. AAAI.

[EGLT76] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–632, November 1976.

[For81] C. L. Forgy. Ops5 user's manual. Technical Report CMU-CS-81-135, Carnegie-Mellon University, 1981.

[For82] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[GM83] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[Gup86] A. Gupta. *Parallelism in Production Systems.* PhD thesis, Carnegie Mellon University, Department of Computer Science, March 1986. Technical Report CMU-CS-86-122.

[HL88] K. E. Huff and V. R. Lesser. A plan-based intelligent assistant that supports the software development process. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 97–106, Boston, MA, November 1988. ACM Press. Special issue of *SIGPLAN Notices*, 24(2), February 1989 and of *Software Engineering Notes*, 13(5), November 1988.

[KBFS88] G. E. Kaiser, N. S. Barghouti, P. H. Feiler, and R. W. Schwanke. Database support for knowledge-based engineering environments. *IEEE Expert*, 3(2):18–32, Summer 1988.

[KBS90] G. E. Kaiser, N. S. Barghouti, and M. H. Sokolsky. Preliminary experience with process modeling in the marvel software development environment kernel. In *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 131–140, Kona HI, January 1990.

[KFP88] G. E. Kaiser, P. H. Feiler, and S. S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49, May 1988.

[Lyn83] N. A. Lynch. Multilevel atomicity — a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[Mir86] D. P. Miranker. *TREAT: A New and Efficient Match Algrithm for AI Production Systems.* PhD thesis, Columbia University Department of Computer Science, June 1986.

[Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, Cambridge MA, 1985.

[MR88] N. H. Minsky and D. Rozenshtein. A software development environment for law-governed systems. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 65–75, Boston MA, November 1988. ACM Press. Special issue of *SIGPLAN Notices*, 24(2), February 1989 and of *Software Engineering Notes*, 13(5), November 1988.

[Per89] D. Perry, editor. *5th International Software Process Workshop*, Kennebunkport ME, October 1989. ACM Press. In press.

[RB89] C. L. Ramsey and V. R. Basili. An evaluation of expert systems for software engineering management. *IEEE Transactions on Software Engineering*, 15(6):747–759, June 1989.

[Sho76] E. H. Shortliffe. *Computer-Based Medical Consultations: MYCIN.* North-Holland, 1976.

[SKW85] D. R. Smith, G. B. Kotik, and S. J. Westfold. Research on knowledge-based software environments at kestrel institute. *IEEE Transactions on Software Engineering*, SE-11(11):1278–1295, November 1985.

A Distributed Implementation of a Multiple View Integrated Software Development Environment

Chris Marlin

Department of Computer Science
The University of Adelaide
G.P.O. Box 498, Adelaide
South Australia 5001, Australia
Telephone: *+61 8 228 5681*
Internet: *marlin@cs.adelaide.edu.au*

ABSTRACT

The MultiView project at the University of Adelaide is investigating the construction of multiple view integrated software development environments which are implemented in a distributed fashion. From the user's point of view, the MultiView environment provides multiple concurrent views of the software system under construction. This allows support for many of the representations employed by software developers, which range from textual descriptions (such as program listings) to various diagrammatic representations (flowcharts, Nassi-Schneiderman diagrams, and so on). The present MultiView prototype supports two kinds of view as a demonstration of the feasibility of the approach.

The implementation of the MultiView integrated software development environment is distributed and is designed to exploit the kind of coarse-grained parallelism to be found in multiprocessor workstations. This style of implementation is motivated by the desire to improve the performance of sophisticated software development environments, particularly with regard to tasks such as incremental code generation. Experience with other kinds of computer-aided engineering environments (say, those for assisting with circuit design) has shown that, as the environments become more sophisticated, the most expensive resource – the engineer – is forced to remain idle for longer periods of time, waiting for response from the environment. The paradigm used to exploit parallelism in the MultiView implementation appears to be applicable to a wide range of computer-aided engineering environments.

1. INTRODUCTION

Although the kind of software development environment used by most software developers consists of a collection of separate tools which have to be repeatedly invoked during the development process, there has been a great deal of progress towards integrated software development environments. These latter systems attempt in various ways to provide environments which are specific to aspects of the software life-cycle.

A particular focus of the work on integrated software development systems has been support for coding. Typical of the results of this research are integrated programming environments such as the Cornell Program Synthesizer [20,21], environments generated by the Synthesizer Generator [17,18] and those generated by the Gandalf system [13]. These systems provide highly specific and integrated support for the development of programs, allowing the manipulation of a program in a language-specific manner and thus facilitating the coding phase of the software life-cycle.

However, most integrated programming environments provide access to the program under development via a single kind of representation. For all of those listed above, the representation used is some pretty-printed form of the source text; this focus on a textual representation appears to be partly the result of the early origins of some of the systems concerned and partly an artifact of the desire to generate the environment from a description of the language to be supported.

With the advent of cheaper workstations with high-resolution displays, some experimental systems have been developed which make use of graphical depictions of programs; for examples of such systems, see [1,6,14]. However, once again, the programmer is usually provided with only one kind of program representation.

The *principal motivation for the development* of the MultiView software development environment at the University of Adelaide has been the observation that developers tend to make use of various representations during software development. As with the work referred to above, our work has also concentrated on the coding phase of the software life-cycle. The representations used in this phase of software development cover both the program specification, which we will refer to as the *static program*, and the program in execution, which is referred to as the *dynamic program*. Typical program representations include:

- textual representations of the static program, such as source listings and text editor displays,
- tree representations of the static program, such as parse trees and trees depicting the nesting of subprogram definitions,
- other diagrammatic representations of the static program, such as flow-charts and Nassi-Schneiderman diagrams, and
- various representations of the dynamic program, such as stacks of subprogram instances and graph structures representing data structures composed of dynamic vari-

ables linked by pointers.

It is also likely that programmers would also find other program representations useful if they were generated automatically by a software development environment.

As noted previously, most existing integrated programming environments provide only one kind of program representation, despite the fact that programmers use the diverse representations listed above. In view of this tendency to use multiple program representations, the MultiView environment aims to facilitate programming by presenting the user with a variety of views of the system under development. We believe that an important principle is that if a view is useful as a depiction of (some aspect of) a program, then it should also be possible to use that same representation as a means of manipulating the program. That is, views should permit editing wherever possible, rather than restricting the programmer to merely employing the view to inspect the program. This principle appears to apply equally well to representations of the static program and to those of the dynamic program.

Consistent with the above observation about programmers' use of multiple representations, some views are planned to present information textually, but others contain graphical representations of the program. For a given object, say a compilation unit in the static program, the user may choose to display several representations of the object. These multiple views of the object are all updated simultaneously whenever one of the views is used to make changes to the object. Among existing integrated programming environments, only PECAN [15,16] appears to include a similar notion of multiple concurrent views; the MultiView work was begun independently of PECAN and the significant differences between the two systems will be mentioned where relevant.

The work on the MultiView environment has been carried out in close association with the development of the Leopard multiprocessor workstation [4,5] at the University of Adelaide. One of the goals of the MultiView implementation has been, therefore, to exploit the kind of coarse-grained parallelism to be found in closely-coupled multiprocessors such as the Leopard. The nature of the MultiView implementation and how it exploits hardware parallelism will be described later.

So far, two prototype implementations of MultiView have been constructed:

- The first, described in [12], was written in Lisp and was a programming environment for a subset of Ada.
- The second prototype, an early version of which is described in [3], is described in some detail in the next section.

A third prototype is currently being implemented and some aspects of this prototype are described in Section 3. Finally, Section 4 discusses some of the future directions anticipated for the work on MultiView.

2. THE PRESENT MULTIVIEW PROTOTYPE

2.1 From the user's point of view

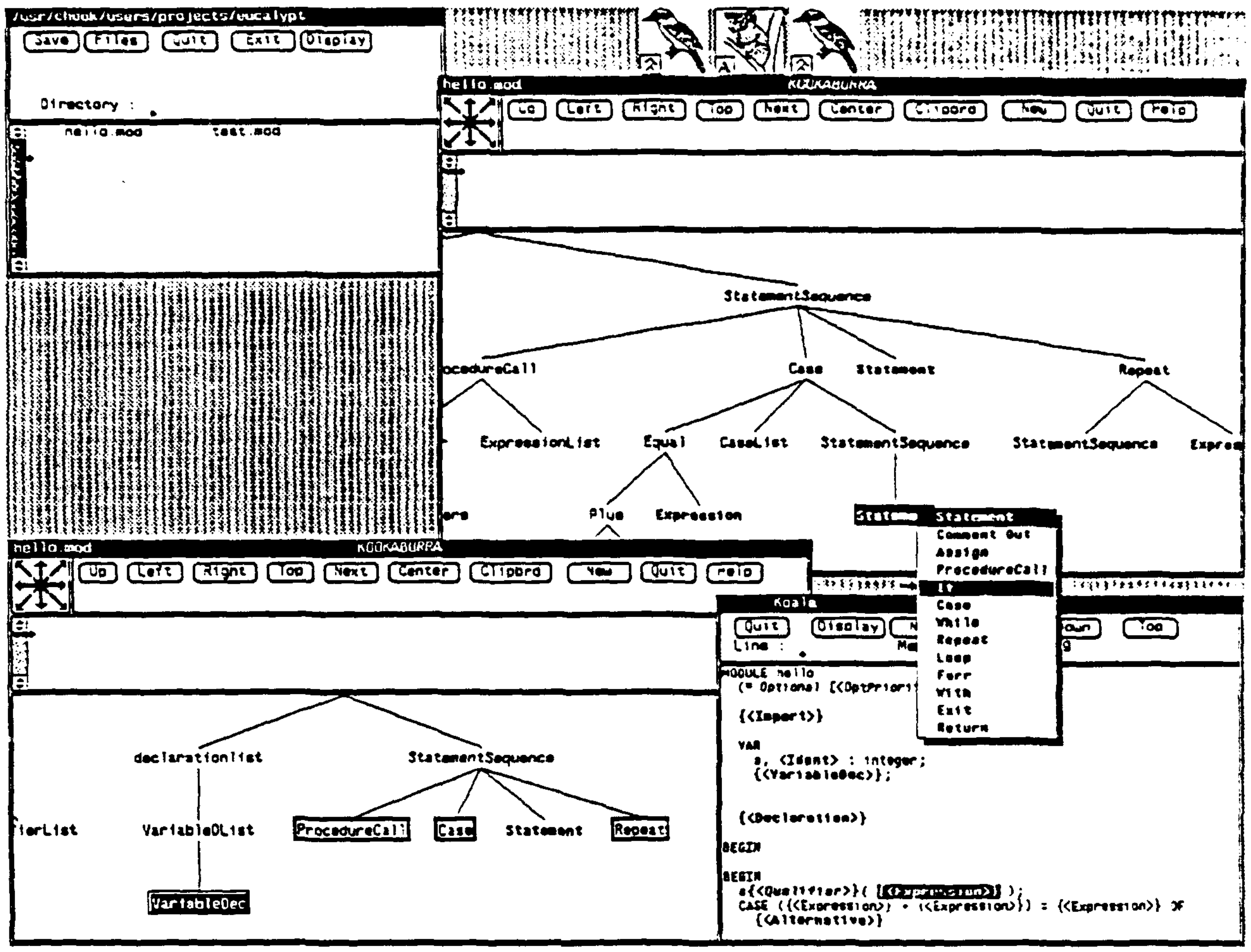
The present status of the MultiView software development environment is illustrated in Figure 1, which shows a session with the system on a Sun workstation. As shown in the figure, the user of the system has the ability to simultaneously display a number of views of the compilation units within the software system being developed. There are two versions of this prototype: one to support software development in Modula-2 (known as Version 1M) and one to support Ada programming (Version 1A). Version 1M, which will be used for the purposes of illustration throughout this section, was developed at the University of Adelaide; Version 1A was derived from it by the Software Engineering Group within the Information Technology Division of the Defence Science and Technology Organization. The differences between the two versions, which are both written largely in Modula-2, essentially amount to a collection of modules describing the language to be supported by the environment.

There are three kinds of window represented in Figure 1: one is called Eucalypt and relates to the management of the compilation units currently under the control of this MultiView session, and the other two (called Kookaburra and Koala) provide a view of a single compilation unit in each case.

As will be seen later in more detail, the essence of the MultiView implementation is to employ a single canonical representation of any compilation unit available to the MultiView system and to derive any required visible representations from this canonical representation. Before discussing the windows in Figure 1 further, it is necessary to briefly discuss the nature of the canonical representation. The representation chosen for compilation units in MultiView is that which is most commonly used to represent program components in language-directed tools: the abstract syntax tree.

An example of a fragment of an abstract syntax tree is depicted in Figure 2. This fragment shows that the elements of the concrete syntax (such as keywords and punctuation) are not present in the tree; these belong to a particular visible representation of the fragment concerned and will be generated if necessary. Adopting the terminology used in the Mentor system [7,8], each node in the tree is said to co¬sist of two parts: the *phylum* and the *operator*. The phylum is an indicator of the syntactic category and covers a range of possible operators. The fragment in Figure 2, for example, has phylum "statement" and this covers the various possible statement types in Modula-2; in the case of this particular tree, the choice has been made that this is an if-statement and so the operator is "if". At some point in program elaboration, the choice of an operator for a particular phylum may not yet have been made – the phylum is then said to be *unexpanded*; an example of this is shown at the righthand side of Figure 2, where there is an unexpanded statement. If the abstract syntax tree in Figure 2 were to be displayed as text, the result would be something of the form:

```
IF X>0 THEN P ELSE (*Statement*) END
```



```
MODULE hello
  (* Optional [<OptPriori
  {<Import>}
  VAR
    a, <Ident> : integer;
    {<VariableDec>};

  {<Declaration>}
BEGIN
BEGIN
  a{<Qualifier>}( [<Expression>] );
  CASE ({<Expression>} + {<Expression>}) = {<Expression>} OF
    {<Alternative>}
```

Figure 1. A session with the present MultiView prototype.

Returning to Figure 1, an instance of the Eucalypt window is shown in the top lefthand corner. A user normally employs a single instance of Eucalypt and uses this to manage the collection of compilation units on which view instances (which will be explained below) may operate; the collection of compilation units shown in the Eucalypt window are all stored in abstract syntax tree form. Apart from showing these compilation units, Eucalypt allows compilation units to be mapped back and forth to the file system, where they can be stored either as source text or in a compressed "database" form. It is also Eucalypt which is used to quit MultiView at the end of a session.

One of the two kinds of view supported in the present MultiView prototype is the Kookaburra view. This view depicts the compilation unit as a tree structure and there are two instances of this view in Figure 1: one in the lower left corner and the other on the upper righthand side. Another example of a Kookaburra view instance is shown in Figure 3. The

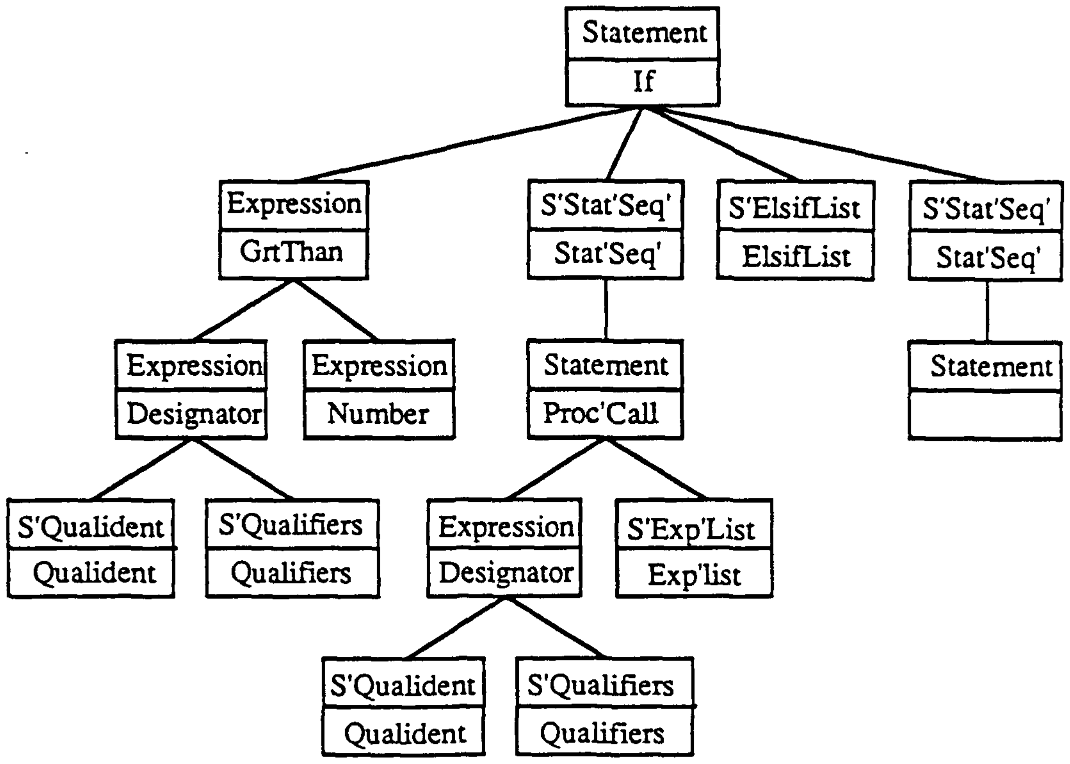


Figure 2. An example of an abstract syntax tree.

operation of the Kookaburra view will not be described in detail – the interested reader is referred to [2] for further details. The Kookaburra view is a simple template-based graphical editor showing the abstract syntax tree. It includes menu-driven expansion of constructs, explicit elision (elided subtrees are depicted by their root node being enclosed in a double border), and other tree-oriented commands. Various conventions are used to display the information in abstract syntax trees such as the one in Figure 2; for example, unexpanded phyla are depicted by the phylum name, the operator name is shown for expanded phyla, and terminal symbols such as identifiers are underlined. A node can be selected as the current node, in which case it is displayed in inverse video and is then subject to commands for deleting subtrees, controlling elision and copying back and forth between the clipboard.

The other kind of view provided in the present MultiView prototype is a text-oriented view. This view is called Koala and is described in detail in [10]. It provides what amounts to a textual language-oriented editor and was designed to provide facilities which are typical of such editors; this design was carried out after the survey of textual language-oriented editors described in [11]. An example of the Koala view is to be found in the lower righthand corner of Figure 1 and another example is shown in Figure 4. The textual representation of the compilation unit

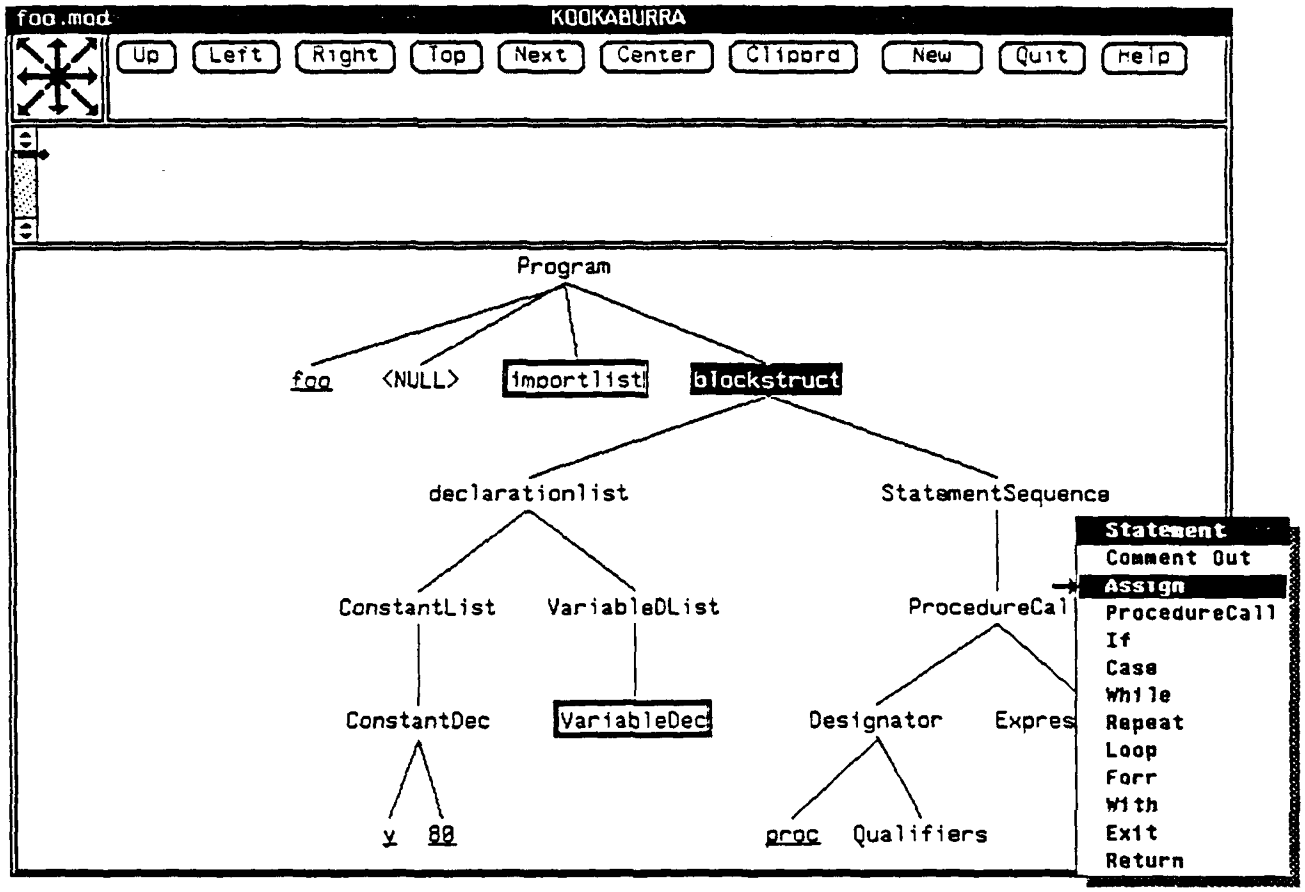
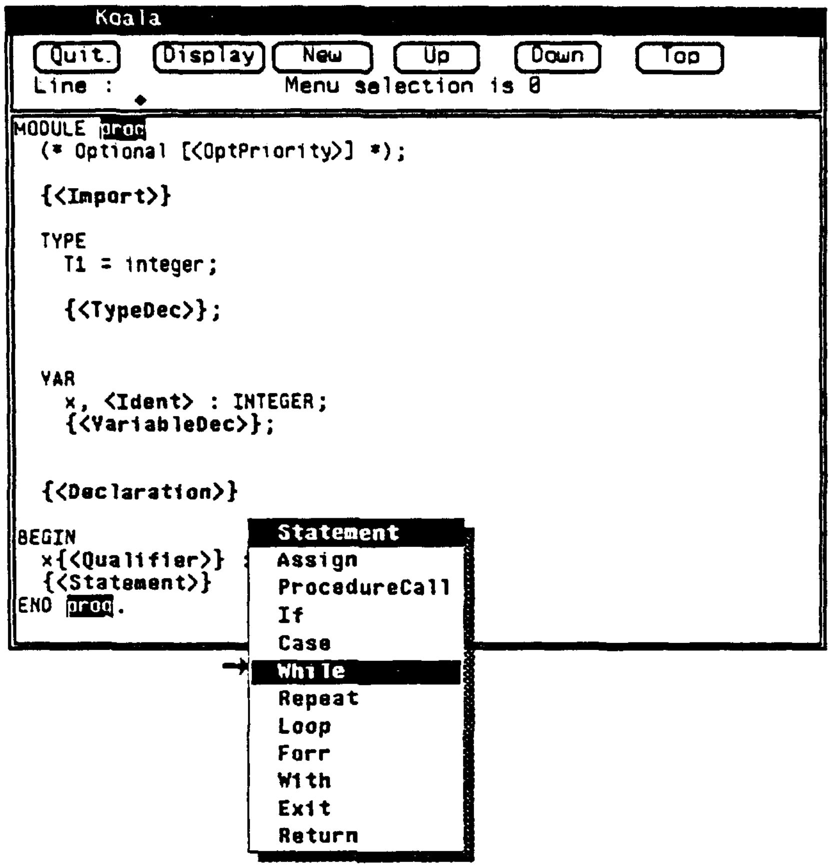


Figure 3. *An example of the Kookaburra view.*

depicted in a Koala view instance is obtained by unparsing the canonical abstract syntax tree representation using different unparsing schemes to those used by Kookaburra. The unparsing schemes used by Kookaburra are language-independent, whereas those for the Koala view must be provided for each new language to be supported.

Given that a user of a multiple view environment such as MultiView will typically be faced at any one time by a collection of view instances drawn from a number of view kinds, it is important that the operation of the various kinds of views be as consistent as possible. This principle has been applied in the MultiView user interface design and so, for example, the template-driven expansion of constructs in Koala is very similar to that in Kookaburra (compare, for example, the two menus in Figures 3 and 4).

Thus, from the user's point of view, the present MultiView prototype provides multiple instances of two kinds of editing views. A change made to a compilation unit in one view instance is immediately propagated to other view instances associated with the compilation unit and the display updated if necessary. The manner in which such a change is propagated is



```
Line :                 Menu selection is 8
MODULE proc
  (* Optional [<OptPriority>] *);

  {<Import>}

  TYPE
    T1 = integer;

    {<TypeDec>};


  VAR
    x, <Ident> : INTEGER;
    {<VariableDec>};


  {<Declaration>}

BEGIN
  x{<Qualifier>} :
  {<Statement>}
END proc.
```

Figure 4. An example of the Koala view.

outlined in the following subsection.

The collection of available kinds of view is clearly very limited at present and is in contrast, for example, to the rich collection of views available in PECAN. This is a reflection of a conscious decision to develop only sufficient view kinds at this stage to demonstrate the feasibility of the MultiView approach, particularly with regard to the implementation structure described in the next section.

2.2 The implementation

The present MultiView implementation runs on a network of Sun workstations and uses the SunView† window management system [19]. The software architecture of the MultiView implementation is outlined in Figure 5: the implementation consists of a collection of concurrently executing processes communicating via message-passing, which has been implemented in terms of UNIX‡ sockets. The exploitation of parallelism that occurs within the MultiView

† SunView is a trademark of Sun Microsystems, Inc.

‡ UNIX is a trademark of AT&T.

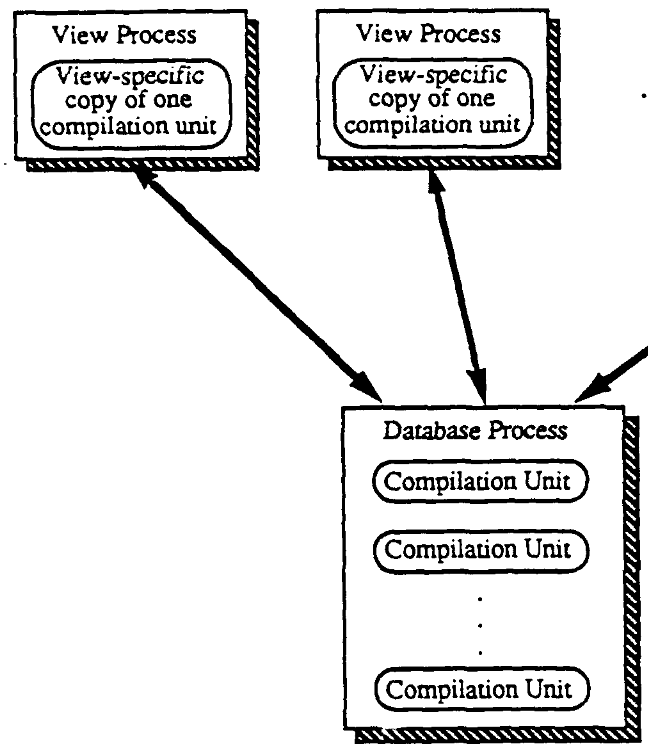

View Process
View-specific copy of one compilation unit
View Process
View-specific copy of one compilation unit
...
View Process
View-specific copy of one compilation unit
Database Process
Compilation Unit
Compilation Unit
Compilation Unit

Figure 5. The software architecture for the MultiView implementation.

implementation depends on the concurrent execution of these processes.

At the heart of the implementation is the database, controlled by the *database process* shown at the bottom of Figure 5. This database holds abstract syntax trees for the collection of compilation units currently being operated upon by the user – that is, those displayed in the corresponding Eucalypt view. The abstract syntax trees held with the database are of the basic kind shown earlier in Figure 2. The database process receives notification of changes to the compilation units held within the database and broadcasts information about modifications when necessary.

Each view instance is managed by a *view process*; if the user is employing a view instance to perform editing on a compilation unit, the corresponding view process receives and interprets the input from the user. Every view process holds a view-specific copy of a single compilation unit; this is an abstract syntax tree which is decorated with information relating to the particular kind of view (such as the coordinates and sizes of the nodes in a Kookaburra view instance). When the user input has been interpreted, the corresponding modifications are made to the local abstract syntax tree and the visible representation updated. Once this has been done, an appropriate description of the changes desired are sent to the database process, which then broadcasts notification to any other view processes holding a copy of the relevant abstract syntax tree; these view processes then update their abstract syntax trees and visible representations.

The interface between view processes and the database process consists of a protocol which is common to all kinds of view process. This protocol, which is outlined in more detail in [3], is independent of the kind of view concerned because it deals entirely in terms of the abstract syntax tree operations. This view-independent protocol has a number of advantages, not least of which is that it is a relatively easy matter to add additional kinds of views: as long as the new view process adheres to the protocol, it is possible to have a new kind of view working in a limited manner with only a few hours work. In fact, we have defined a "skeleton" view process which provides the communication facilities and a basic abstract syntax tree.

The implementation structure depicted in Figure 5 allows view instances to be updated in parallel, which contrasts with the round-robin scheduled updating of views which occurs in PECAN. However, as mentioned above, the present prototype implementation of MultiView uses the SunView window management system. This has the unfortunate effect that all of the view processes must be run on the same processor: that belonging to the Sun workstation which owns the frame-buffer on which the view instances are being displayed. This fact was a major motivating factor in our decision to use the X Window System[†] for the version of MultiView outlined in the next section. The present prototype does, however, permit the database process to be run remotely from the view processes and this improves the performance to some extent.

Another limitation on the parallelism in the present prototype is that requests to access the database are handled serially – a new request will not be taken until the previous one has completed. Various ways of relaxing this restriction on parallelism are briefly outlined later. This prototype also does not include semantic analysis, code generation or execution; these are planned for the version of MultiView discussed below and the exploitation of parallelism is a major goal in their design.

3. CURRENT WORK

As indicated in the previous section, the present MultiView prototype uses the SunView window management system and is largely implemented in Modula-2 (the remainder provides the interfaces to SunView and is written in C). The current work is focussed on rewriting the MultiView implementation in Ada; the X Window System will be used to provide window management in this new prototype. The decision to embark on this re-engineering of the MultiView implementation was based on a number of factors, including:

- the assessment that the present prototype was not sufficiently well engineered to provide a good basis for the planned work on semantic analysis, code generation and run-time views, and
- the previously mentioned difficulties of obtaining true parallel execution of view pro-

[†] X Window System is a trademark of the Massachusetts Institute of Technology.

cesses under the SunView system.

Initially, the work is aimed at providing another environment to support the development of systems in the Modula-2 language; this version is called Version 2M. A version to support programming in Ada will follow in due course.

The software architecture of this new prototype is essentially the same as that shown previously in Figure 5. Experience with the present prototype has indicated that the communications between the processes are crucial to both the correct operation and the performance of MultiView. Consequently, some considerable effort has been devoted to the redesigning of this aspect of the MultiView implementation to introduce greater redundancy in the communications protocol. Currently, the database process and the communications interface between the database process and view processes have been largely completed.

As indicated, MultiView is also being modified to make use of the X Window System instead of SunView system to carry out the user interaction. The first kind of view being developed for the new MultiView prototype is a text-oriented view similar to the Koala view described in the previous section. The workstation with the display screen being used for user interaction now runs as an X-server, permitting all view processes and the database process to be run on other workstations and servers. This will significantly increase the amount of concurrency possible in the implementation. More distribution of the processes in the MultiView implementation will not only improve its performance, but will also help to prepare for the ultimate goal of running the system on a multiprocessor workstation.

4. CONCLUSIONS AND FUTURE WORK

A multiple view integrated software development environment and its distributed implementation have been outlined. Two prototype editing environments based on these ideas have been constructed and a third is currently being constructed. Experience with the completed prototypes, particularly the more sophisticated second prototype described in Section 2, suggests that there are many potential benefits to be gained from providing multiple concurrent views which better support the multiplicity of ways that a software developer views the software system being constructed. However, it is equally clear that users vary significantly with respect to the kinds of views that they find useful and the manner in which they use them. This has a number of ramifications, including the following:

- it will be necessary to offer the user a large variety of view kinds from which to choose, and
- customization will be important – a user will want to express preferences about the view kinds to be available and the view instances to be created when returning to work on some part of a system or when beginning work on some new task.

It is also clear that a great deal more work is yet to be done on the kinds of views to be provided. Apart from looking to existing representations (such as flow-charts and Nassi-

Schneidermann diagrams) for inspiration about useful view kinds, it will also be necessary to develop new kinds of view which are only made feasible by the fact that they are being automatically, rather than manually, generated. This is particularly true in the area of run-time views and related debugging aids.

Each of the MultiView prototypes has been written with automatic generation from language descriptions in mind. However, we have never allowed the current state of the art in automatic generation tools to limit the design of the system, say with regard to the kinds of view provided. Thus, as mentioned earlier, there are aspects of the MultiView implementation which are generated from the language to be supported by the environment, such as repositories of information about the abstract syntax of the language and the textual form of constructs; this information is then used in displaying menus, presenting an abstract syntax tree in textual form, and so on. Other aspects of the implementation may have to be modified manually when creating a MultiView implementation for a new language.

Once Version 2M of MultiView has reached the same external appearance as that described for Version 1M in Section 2 (that is, with a textual view and a graphical view), it will then be used as a platform for the introduction of semantic analysis and code generation. It is planned to use some form of incremental attribute evaluation for carrying out the semantic analysis and it is unlikely that this will be carried out in parallel with further editing – to do so would potentially lead to confusing effects from the user's point of view, where semantic errors were being uncovered away from the user's present focus, and is not likely to significantly improve performance.

With regard to code generation, however, there appear to be considerable possible benefits to employing parallelism. One approach to exploiting parallelism in incremental code generation has been demonstrated in the implementation of the PSEP programming environment [9]; in this system, a "customized" form of concurrency control is used, making use of specialized information known about the collection of processes involved, and a similar approach is being evaluated for MultiView.

Following the introduction of code generation, program execution will be implemented and this will give an opportunity to introduce some dynamic views. Prototyping of dynamic views has already begun at the Defence Science and Technology Organization, where Version 1A of MultiView is being used as a front-end to a traditional Ada implementation. The Ada compilation units prepared by MultiView can be submitted to an Ada compiler running on a remote machine; the resulting code will then be executed there (or elsewhere) under the control of processes implementing run-time view instances on the user's workstation and communicating with the executing program via message-passing.

At present, the MultiView implementation runs in a distributed fashion over a local area network. Eventually, we plan to move MultiView from this loosely-coupled environment to the closely-coupled architecture of the Leopard multiprocessor workstation mentioned in Section 1. The processes of the MultiView implementation would then be distributed over the closely-coupled multiprocessor in some manner, or perhaps over a collection of such multiprocessors connected in a local area network. The Leopard will provide a suitable platform on which to test the effectiveness of the multiple process architecture of MultiView in exploiting the parallelism of multiprocessor workstations to provide efficient implementations of sophisticated software engineering environments. The software architecture employed in the MultiView implementation also appears to provide a suitable basis for the distributed implementation of systems in areas as diverse as computer-aided design and office automation.

The MultiView environment described in this paper attempts only to provide support for a single programmer working on the coding aspects of some software project. Future plans include support for other aspects of the software life-cycle and will necessarily involve the co-ordination of the work of many individuals. Once again, examination of practices in these other aspects also reveals many instances of the use of multiple representations; for example, project management frequently involves the generation and consultation of various representations of information relating to task assignments, progress towards deadlines, and so on. There would also appear to be many opportunities to incorporate knowledge-based components in a multiple view environment which is attempting to provide complete life-cycle support and is based on our multiple process software architecture; our future plans also include examination of some of these opportunities.

ACKNOWLEDGEMENTS

A large number of individuals have made contributions to the MultiView project since its inception in 1985. These contributors are too numerous to list exhaustively here; however, the contributions of Richard Altmann, Andrew Hawke, Michael McCarthy, Neville Parker. Keith Ransom and Bradley Schmerl are worthy of special mention. The cooperation and financial assistance of the Defence Science and Technology Organization are gratefully acknowledged; financial support has also been provided by the Australian Research Grants Scheme, the Australian Research Council, the Australian Computer Research Board, the Australian Telecommunications and Electronics Research Board, and the University of Adelaide.

REFERENCES

[1] M. B. Albizuri-Romero, "GRASE – A Graphical Syntax-Directed Editor for Structured Programming", *A.C.M. SIGPLAN Notices*, Vol. 19, No. 2 (February 1984), pp. 28–37.

[2] R. A. Altmann, "An Abstract Syntax Tree Editor for the MultiView Programming Environment", Honours project report, Department of Computer Science, The University of Adelaide, Adelaide, South Australia (October 1986).

[3] R. A. Altmann, A. N. Hawke and C. D. Marlin, "An Integrated Programming Environment Based on Multiple Concurrent Views", *Australian Computer Journal*, Vol. 20, No. 2 (May 1988), pp. 65–72.

[4] P. J. Ashenden, C. J. Barter and C. D. Marlin, "The Leopard Workstation Project", *A.C.M. Computer Architecture News*, Vol. 15, No. 4 (September 1987), pp. 40–51.

[5] P. J. Ashenden, C. J. Barter and M. A. Petty, "The Leopard Multiprocessor Workstation Project", Technical Report LW-1, Centre for Computer Systems and Software Engineering, The University of Adelaide, Adelaide, South Australia (November 1989).

[6] S.-K. Chang (Ed.), *Principles of Visual Programming Systems* (Prentice-Hall, Englewood Cliffs, New Jersey, 1990).

[7] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, "Programming Environments Based on Structured Editors: The MENTOR Experience", Technical Report No. 26, Institut National de Recherche en Informatique et en Automatique (INRIA), Rocquencourt, Le Chesnay, France (1980).

[8] V. Donzeau-Gouge, G. Kahn, B. Lang, B. Mélèse and E. Morcos, "Outline of a Tool for Document Manipulation", *Information Processing 83*, R. E. A. Mason (Ed.), pp. 615- 620.

[9] R. Ford and D. Sawamiphakdi, "A Greedy Concurrent Approach to Incremental Code Generation", *Conf. Record of the Twelfth Annual A.C.M. Symp. on Principles of Programming Languages*, pp. 165–178.

[10] C. S. Lee, "A Textual View for the MultiView Programming Environment", Honours project report, Department of Computer Science, The University of Adelaide, Adelaide, South Australia (October 1987).

[11] C. D. Marlin, "Language-Specific Editors for Block-Structured Programming Languages", *Australian Computer Journal*, Vol. 18, No. 2 (May 1986), pp. 46–54.

[12] M. J. McCarthy, "Towards an Integrated Programming Environment Based on Multiple Concurrent Processes", Honours project report, Department of Computer Science, The University of Adelaide, Adelaide, South Australia (November 1985).

[13] D. Notkin, "The GANDALF Project", *J. Systems and Software*, Vol. 5, No. 2 (May 1985), pp. 91–105.

[13] G. Raeder, "A Survey of Graphical Programming Techniques", *I.E.E.E. Computer*, Vol. 18, No. 8 (August 1985), pp. 11–25.

[15] S. P. Reiss, "Graphical Program Development with PECAN Program Development Systems", *Proc. A.C.M. SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* {*A.C.M. SIGPLAN Notices*, Vol. 19, No. 5 (May 1984)}, P. Henderson (Ed.), pp. 30–41.

[16] S. P. Reiss, "PECAN: Program Development Systems that Support Multiple Views", *I.E.E.E. Transactions on Software Engineering*, Vol. SE-11, No. 3 (March 1985), pp. 276–285.

[17] T. W. Reps and T. Teitelbaum, *The Synthesizer Generator Reference Manual* [Third Edition] (Springer-Verlag, New York, 1989).

[18] T. W. Reps and T. Teitelbaum, *The Synthesizer Generator: A System for Constructing Language-Based Editors* (Springer-Verlag, New York, 1989).

[19] Sun Microsystems, "SunView Programmer's Guide", Part Number 800-1345-10, Sun Microsystems, Mountain View, California, September 1986.

[20] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Commun. A.C.M.*, Vol. 24, No. 9 (Sept. 1981), pp. 563–573.

[21] T. Teitelbaum, T. Reps and S. Horwitz, "The Why and Wherefore of the Cornell Program Synthesizer", *Proc. A.C.M. SIGPLAN/SIGOA Symposium on Text Manipulation* {*A.C.M. SIGPLAN Notices*, Vol. 16, No. 6 (June 1981)}, pp. 8–16.

SPECIFICATION EVOLUTION AND PROGRAM (RE)TRANSFORMATION

Martin S. Feather[1]

USC / Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
(213) 822-1511

Program transformation has been proposed as a paradigm for developing efficient programs from specifications. Human selected, machine applied transformations preserve the meaning of a specification while improving its efficiency. Crucial to the success of this paradigm is the ability to *replay* a specification's transformational development when that specification is changed, in order to reduce the cost of such re-development.

We have extended the paradigm through the use of 'evolution' transformations that deliberately *change* the meaning of the specifications to which they are applied. Using these, the relationship between a changed specification and its original form is captured in the record of the evolution transformations applied to make that change. This information appears useful for alleviating some of the problems that arise during replay of transformational developments, and complements other approaches to making developments replayable.

We demonstrate these ideas on a simple example, showing how several separate changes can be captured as invocations of evolution transformations and then combined by serial replay of those transformations.

1. Program Transformation and Replaying Developments

The transformational paradigm for program development suggests that efficient programs be derived from lucid specifications by the application of meaning-preserving transfor-

[1]This research has been supported in part by Defense Advanced Research Projects Agency grant No. NCC-2-520, and in part by Rome Air Development Center contract No. F30602-85-C-0221 and F30602-89-C-0103. Views and conclusions contained in this document are those of the author and should not be interpreted as representing the official opinion or policy of DARPA, RADC, NSF, the U.S. Government, or any other person or agency connected with them. The author wishes to acknowledge the Software Sciences Division at ISI for providing the context in which this research has been conducted, and in particular Kevin Benner for his insightful comments on an earlier draft of this paper.

mations (e.g., [1, 2]). If transformation is completely automatic (i.e., compilation), it requires the compiler to find a tolerably efficient implementation of the specification. Since this becomes more difficult as the specification language becomes more expressive (and so further removed from efficiency concerns), human guidance is often sought to help guide the transformation process, while retaining the use of a mechanical system to conduct the chosen transformations (such systems are surveyed in [15]).

Within this paradigm, program maintenance is achieved by changing the specification and re-developing the efficient program from that changed specification. It is thus important to make use of the transformational development of the original specification during re-development from a changed specification, in particular, to minimize the need to seek human assistance again. This is particularly crucial if transformational development from the original specification is harder than simply writing the program directly. In such a case, the paradigm may still realize a net saving if the effort of the initial development can be amortized over the lifetime of the program, so that the maintenance cycle (i.e., change specification and re-develop program) is far cheaper and easier via transformation replay than via changing (patching) the program directly[2]. For human guided transformation, the guidance is typically captured as a structured record of the transformation invocations [18, 6]. Once constructed, such a record can be automatically executed by the transformation system on its input (a specification) to produce its output (a program). Ideally, it should be possible to replay a development on modified specification so as to automatically re-derive the program [5]. Unfortunately it appears that transformational developments are not very robust — they do not replay well.

In the next section we outline existing approaches toward the goal of increasing the robustness (replayability) of transformational developments, and introduce our approach, which involves extending the transformational paradigm. Then, in section 3 we illustrate our approach by applying it retroactively to a portion of an already fielded system. Finally, in section 4, we discuss how our approach has indeed facilitated replay, with reference to the previous section's example for illustration.

2. Robust (Replayable) Transformational Developments

We outline several approaches to making mechanically supported transformational developments robust (replayable), the last of which is the novel approach that we are proposing.

Layered developments

The transformational development is structured into 'layers', so that each layer comprises

[2]Proponents of program transformation argue that cost-saving is but one of several potential benefits, others including increased assurance that the program is a correct implementation, and better understanding of how the program works.

a translation from one language level to another — this approach has been applied in Boyle's Lisp to Fortran transformations, [4], and in the compiler for the high-level language Refine [17]. This has the advantages of clearly characterizing the purpose of each layer (to move to the next language level), and permitting completely automatic replay provided that the modified specification is still expressed entirely within the topmost layer's language.

Heuristics
'Heuristics' that operate upon the transformational development can be used to enhance its replayability. For example, Goldberg uses heuristics to relate portions of the old and new specification based upon name and structure similarities, and correspondencies between outputs of corresponding transformation applications [10]. Heuristics can also be used to point out weaknesses of the existing development. For example, Mostow proposes a heuristic to detect when the rationale recorded with a transformation step is incomplete (i.e., the transformation might not work in all the circumstances that the rationale claims it should) [13].

Recording rationale
The 'rationale' behind a transformational development is recorded, e.g., why one transformation was chosen rather than another. Since this record will capture more of the general strategy underlying the transformational development, it should prove more robust in the face of specification change than would the record of the transformation applications alone. The DEVA project is working on formalisms specifically for expressing this information, and is accumulating examples of developments [16]. A mechanization of the rationale-based approach has been studied by Fickas, who cast the problem as one of 'planning', in which the goal structure of the process is made explicit and manipulated (transformation invocations serve as the operators in this planning space, selection criteria choose among transformations, and goals characterize program properties to be sought or eliminated) [9].

Our approach — evolution transformations
Our approach is based on the thesis is that it is advantageous to know *how* the specification has changed when we need to replay the transformational development. In order to capture this information, we employ so-called 'evolution transformations' to *change* the meaning of specifications. Unlike conventional program transformations, these are not restricted to only preserving or diminishing (i.e., making implementation choices from 'don't care' alternatives) the meaning of specifications. Further motivation for, and examples of, our approach to using evolution transformations can be found in [7]. Here we focus on their implications for replay of transformational developments.

We give equal status to the transformations used for evolution and the transformations used for optimization, and view retransformation as the combination of all the transformations. We will use serial replay of all the transformations, both evolutionary and meaning preserving, to achieve this combination, but prior to doing so we compare the transformation applications themselves so as to predict (and forestall) some of the complications that arise during replay.

3. Example of Specification Evolution and Replay

We show our approach in action by describing a simple portion of a system, making several changes to that portion independently, achieving each of those changes by transformation, and finally combining all the changes by serially replaying all of the transformations. We demonstrate how a mechanical comparison of the transformation applications predicts interactions among them (leading to a choice of alternative combinations) and how the designer may easily select the appropriate choice and achieve this through mechanized replay.

3.1. Informal Description of Parsing Example

Our example is drawn from Wile's POPART system [18], a grammar-driven tool generator that we use extensively. We focus on a small but central aspect of the system, namely *parsing*. Over the years, Wile has manually changed POPART's parser to enhance its functionality and improve its efficiency. We begin with informal descriptions of the initial state of the parsing portion and the changes made to it. In the following sections we show how to specify the portion formally (section 3.2), how to achieve each of the changes by transformation (section 3.3), and how to combine all the changes by serially replaying those transformations (section 3.4).

The basic activity of POPART's parsing function is very simple - it is given a string and a grammar, and returns either a parse tree (the parse of that string with respect to that grammar) or an indication of failure (if there is no parse). This parse function is called from many places by the rest of the system. The changes that we will be concerned with all leave the calls to the parse function unchanged, but alter its behavior. A brief description of each of these changes follows next.

Pre-filtering change: some strings are to be rejected as having no parse, regardless of whether the original parsing function would successfully parse them.

Post-filtering change: some otherwise successful parses are to be rejected or their resultant parse trees modified. This is not quite symmetrical to pre-filtering, because here the post-filter operation can not only reject but also modify parses.

Compaction change: the parse trees produced by successful parsing of strings are to be modified by 'compacting' them, i.e., making them smaller by discarding redundant information.

Rapid Rejection change: a simple test - much cheaper than calling the full-blown parsing function - can rapidly detect that some strings cannot be parsed. This is introduced as an optimization, on the understanding that the savings from not having to call the parser on the many inputs that it rejects exceed the extra time to perform this test on all inputs.

Additional Parses change: an additional class of parses are introduced, i.e., a second parsing function is added that produces parses of some strings that the original parsing function rejects.

Clearly there is overlap between the above changes, for example, compaction is a special case of post-filtering. Despite these overlaps, we keep these changes separate, mirroring the conceptual separations of their purposes.

3.2. A Formal Specification for Transformation

Wile's code is written in CommonLisp, however we do not transform this code directly. Instead, we re-express the essential details of the problem in a specification language of types, relations, predicates and expressions, and transform that representation. The advantages of such re-expression are that it it allows us to abstract away from the inessential features of the problem (e.g., how parsing is actually done!), and from the encodings implicit in the lisp program (e.g., the representation of a failure to parse as the distinguished value NIL returned by the parse function instead of a parse-tree).

Our first cut at a very simple specification of the relevant aspects of the parsing problem is as follows:

```
{ type string; type parse-tree;
  relation P(s:string,pt:parse-tree) iff IP(s,pt);
  relation IP(s:string,pt:parse-tree) }
```

The above declares two types, `string` and `parse-tree`, and two binary relations, `P` and `IP`. `IP` represents parsing, i.e., it holds between a string `s` and a parse tree `pt` — which we write as `IP(s,pt)` — whenever the parse[3] of `s` is `pt`; since we are not concerned with the actual mechanism of parsing, we need provide no further details of `IP`. `P` is defined in terms of `IP`; in this initial specification, it is defined to be the same (`P(s,pt)` holds if and only if `IP(s,pt)` holds). As we shall see, all of our changes are captured as changes to `P`'s definition.

[3]We have also simplified the problem by omitting any mention of which grammar is to be used for parsing — this could be expressed by an additional parameter of each relation, e.g.,
`relation(s:string, pt:parse-tree, g:grammar) iff IP(s,pt,g)` etc.
This we can also introduce via evolution transformation, but for brevity we will not consider this.

Note that in our proposed initial specification, the purely relational definition of P in terms of IP conveys no notion of inputs or outputs. For our purposes it is better to make these aspects more explicit. We do this by re-expressing P's definition as an equality between the 'output' parse tree pt and the result of querying IP on the 'input' string s, thus:

```
{ type string; type parse-tree;
  relation P(s:string,pt:parse-tree) iff pt = IP(s,?);
  relation IP(s:string,pt:parse-tree)
```

> *Notation:* IP(s,?) is an expression denoting the parse tree that is related to string s by relation IP. For example, writing + (addition) as a three-place relation whose third place is the sum of the first two, +(5,7,?) is an expression equal to 12, +(?,7,12) is an expression equal to 5, etc.

In the remainder we will omit repeating the type declarations in P's definition, writing
 relation P(s,pt) ... instead of
 relation P(s:string,pt:parse-tree) ...

3.3. Realizing Each Change by Transformation

We consider in turn each of the five changes, showing how they are represented on our specification, and how they can be realized through application of an evolution transformation. We have used our transformation system to perform each of these changes mechanically.

Pre-filtering change: Pre-filtering is a test applied to the input string, such that if the string fails the test, then the result is a failure to parse, regardless of whether or not the original parsing function would have been able to parse the string successfully. This can be captured in the specification's definition of P as follows:

```
relation P(s,pt) iff PRE(s) and pt = IP(s,?)
```

where PRE is a unary relation on strings representing the test (i.e., PRE(s) holds if and only if s passes the test).

Thus whereas P was originally defined directly as pt = IP(s,?), now a conjunction with a query of PRE has been interposed. This is an instance of a very general change that we call 'splicing', where some direct connection between two nodes A and B has been changed by the splicing in of an some intermediate node C, thus: $A \longrightarrow B$ becomes $A \longrightarrow C \longrightarrow B$. We have found occurrences of splicing under many guises, depending upon the interpretation of the nodes and links. For example, if nodes A and B are types in a type hierarchy (so that B is a specialization of A), then splicing models the introduction of some intermediate type node C, such that C is a specialization of A, and B is a specialization of C. Splicing is one of several very general operations that we find are ubiquitous in specification change; for a discussion of these operations and how we are using them to organize our transformation library, see [12].

Returning to our pre-filtering change, we employ the splice evolution transformation for nodes of type expression. To invoke it, we must indicate:
where to do the splicing: around the predicate defining `P`, and
the new expression to be spliced in: '`PRE(s) and ♣`', where the symbol '♣' indicates the place into which the original expression is to go. Since the original expression was `pt = IP(s,?)` this splicing results in `PRE(s) and pt = IP(s,?)`.

Post-filtering change: Post-filtering can be captured in `P`'s definition as follows:

```
relation P(s,pt) iff pt = POST(IP(s,?),?)
```

where `POST` is a binary relation between two parse trees such that `POST(pt1,pt2)` holds if `pt1` is not to be rejected by post-filtering, and `pt2` is the post-filtered form of `pt1` (if it is not modified, then `pt1` and `pt2` will be the same). Again, this change can be viewed as 'splicing', but this time between the expression `IP(s,?)` and the right hand side of the '`=`' in `P`. Hence we employ the same splice change transformation as before, but with different input values, namely:
where to do the splicing: around the right hand side of the equality with `pt`, and
the new expression to be spliced in: '`POST(♣,?)`' (using ♣ as a place-marker as before).

Compaction change: As we remarked earlier, compaction is a special case of post-filtering, hence looks very similar when incorporated into the original specification:

```
relation P(s,pt) iff pt = COMP(IP(s,?),?)
```

where `COMP` is a binary relation between two parse trees such that `COMP(pt1,pt2)` holds if `pt2` is the compacted form of `pt1` (since compaction, unlike post-filtering, is not allowed to reject parses that `IP` produced, then every `pt1` that could be in the `IP` relation is related to some `pt2` by the `COMP` relation). To achieve this via invocation of the splice evolution transformation, we give as inputs:
where to do the splicing: around the right hand side of the equality with `pt`, and
the new expression to be spliced in: '`COMP(♣,?)`'.

Rapid Rejection change: Rapid Rejection is similar to pre-filtering, in that the string is subjected to a new test whose failure causes failure to parse. Thus the evolved form of the original specification is:

```
relation P(s,pt) iff (not RR(s)) and pt = IP(s,?)
```

where `RR` is a unary relation on strings representing the test for a rapid rejection (i.e., `RR(s)` holds if and only if `s` is to be rejected). For this to be a pure optimization, it must be the case that for any string related to some parse-tree by the `IP` relation, `RR` must not hold of that string. Invoking the splice evolution transformation to achieve this requires indicating:

where to do the splicing: around the predicate defining P, and
the new expression to be spliced in: '(not RR(s)) and ♣'.

Additional Parses change: Our last change introduces an additional class of parses. Expressed on our specification, this gives:

```
relation P(s,pt) iff pt = IP(s,?) ⊔ AP(s,?)
```

where AP is a binary relation between a string and a parse tree representing the new parse (presumably no string is in both the IP and AP relation), and ⊔ is an infix operator between expressions such that e1 ⊔ e2 is an expression whose referents are the union of the referents of e1 and e2, so pt = A ⊔ B expands to pt = A or pt = B. Invoking the splice evolution transformation to achieve this requires indicating:
where to do the splicing: around the right hand side of the equality with pt, and
the new expression to be spliced in: '♣ ⊔ AP(s,?)'.

3.4. Realizing All the Changes by Transformation Replay

In the previous section we showed how each change could be achieved by the appropriate invocation of an evolution transformation. These were realized independently, i.e., we evolved the initial specification each time, giving rise to a different specification for each change. Now, we want to combine the changes to have a single specification that incorporates them all. As a simple example, consider combining just two of the changes — post-filtering and pre-filtering — the former puts a call to POST around the result returned by IP:

```
relation P(s,pt) iff pt = POST(IP(s,?))
```

while the latter adds a conjunct querying PRE on the input string:

```
relation P(s,pt) iff PRE(s) and pt = IP(s,?)
```

The 'obvious' semantic combination of these is:

```
relation P(s,pt) iff PRE(s) and pt = POST(IP(s,?),?)
```

To achieve this, we *serially* replay the invocations of the evolution transformations.

Serial replay and the 'reference' problem: During serial replay of transformations that were developed independently, problems may arise if transformations replayed later in the serialization contain references to locations in the specification, since those references may not evaluate to the same locations in the specification once it has been changed by the earlier transformations. For example, the location IP(s,?) in the initial specification's iff pt = IP(s,?) could be referenced as the query whose first argument is s, or as the right hand side of the equality. After the evolution step introducing post-filtering, the latter reference no longer evaluates to the same location: post filtering changes the specification to: iff pt = POST(IP(s,?),?) in which the query whose first argument is s remains IP(s,?), whereas the right hand side of the equality is now POST(IP(s,?),?).

This problem, the potential for disturbing references to locations within specifications, has been termed the 'reference' or 'correspondence' problem [13, 10].

In our experiments we dealt with this by canonicalizing the location descriptions of all the transformations to be serially replayed (our canonical form is essentially the path from the top of the parse tree to the location), and ordering the splice transformations so that splices to locations lower in the tree are always performed before splices to locations higher in the tree. In our example this means applying the post-filtering transformation before the pre-filtering transformation (since the former's location is the query of IP within the predicate defining P, whereas the latter's location is the whole predicate defining P). This serial replay gives us our desired result, namely:

```
        relation P(s,pt) iff PRE(s) and pt = POST(IP(s,?),?)
```

We have considered (but not implemented) some other ways to deal with the reference problem:

- Have the earlier transformations change not only the specification, but also the later transformations' references to locations in that specification.
- Use the transformations to maintain correspondencies between locations within the original and changed specifications. For example, splicing introduces an intermediate node into a link, but leaves the other links and nodes unchanged. Thus the correspondence between all unchanged links and nodes could be maintained by the evolution transformation, while the link that has been spliced is decomposed by the transformation into two links. This approach would extend Goldberg's use of heuristics to establish and maintain correspondencies during retransformation.
- Apply some technique other than serial replay of transformations in order to combine the specification versions — such a technique has been developed by Horwitz, Prins and Reps [11] for use when the versions are non interfering. However, as we consider next, there are occasions when the versions do interfere.

Semantic choices during replay:

For some pairs of evolutions, there is a *semantic choice* of how to combine them (we can also imaging cases in which two evolutions are contradictory, i.e., there might be no way to reasonably combine the effects of both). Consider the evolutions of compaction and additional parses: one possibility is to have compaction done on parse trees that result from both the original parsing, IP, and the additional parsing, AP, thus:

```
        relation P(s,pt) iff pt = COMP( IP(s,?) ⊔ AP(s,?) )
```

while the alternative is to have compaction is done on only those parse trees that result from original parsing, thus:

```
        relation P(s,pt) iff pt = COMP(IP(s,?)) ⊔ AP(s,?)
```

We stress that this semantic choice arises as a consequence of trying to combine the effects of both the splice transformations; we need further information to resolve this choice - either alternative gives a viable (but different) program. Thus we see that combination cannot be totally automated, since further input may be needed to resolve choices that arise when evolutions that were conceived of separately are to be combined. However, we can provide automated support, both to detect these cases, and to perform the alternative we select.

Detecting semantic choices:

To detect semantic choice we compare the invocations of the evolution transformations. For example, knowing that the transformations to introduce compaction and additional parses are both *splices* of the *same* link of the original specification, we can predict that there will be a semantic choice of combinations of those splices. This can be seen from considering splicing in the abstract: if one splicing evolution transformation splices in node C between A and B, $A \longrightarrow C \longrightarrow B$ while another splices in node D, $A \longrightarrow D \longrightarrow B$ then the combination of the two could be in either order, $A \longrightarrow C \longrightarrow D \longrightarrow B$ or $A \longrightarrow D \longrightarrow C \longrightarrow B$ (or possibly we may be able to combine C and D in some other way). These alternatives will be distinct unless C and D commute, as would be the case if they were both conjuncts added to B. Conversely, if two splice evolution transformations are applied to different links of the original specification, we would know there is no semantic choice of how to do combination. Note that this reasoning is done at a very abstract level — it would apply to any kind of 'splice', regardless of the nature of the links that were being spliced. This is the logical continuation of the approach outlined in [8], where it was suggested that evolution transformations be characterized by how they affect various aspects of specifications, and these characterizations compared to detect sensitivity to ordering (or other kinds of interference among evolution transformations).

In order to try this on our parsing example, we automated the comparison of invocations of expression splicing, and used this simple mechanism to find the following:

- Post-filtering, Compaction, and Other Parsing are splices of the same link of the original specification. Since we have no reason to suppose that they commute, each of the six possible orderings of these three transformations is a distinct alternative.

- Pre-filtering and Rapid-Rejection are splices of the same link of the original specification, so are candidates for ordering sensitivity. Since both of them splice in an expression by conjunction, we know that they can commute (this last piece of reasoning has *not* yet been built in to our system)[4].

[4]Actually, the Rapid-Rejection change was to realize an efficiency saving by running RR first, so its conjunct should be ordered appropriately.

- The link spliced by evolutions in the first group (Post-Filtering, etc.) is below the link spliced by evolutions in the second group (Pre-Filtering and Rapid-Rejection), so semantically these groups are independent. However, to ensure that the first group's transformations' (canonicalized) references to the lower link's location are not affected by transformations of the second group, all the transformations of the first group are performed before any of the transformations in the second group.

Performing the selected semantic combination:

Whichever semantic combination we desire, we want to somehow achieve that combination. To do this, we simply replay the evolutions in the appropriate order: If we apply the evolution to introduce additional parsing first, then the evolution to introduce compaction second, we get:

```
          relation P(s,pt) iff pt = COMP( IP(s,?) ⊔ AP(s,?) )
```

In the other order, we get:

```
          relation P(s,pt) iff pt = COMP(IP(s,?)) ⊔ AP(s,?)
```

Thus by suitably ordering the splice evolution transformations that apply to the *same* link, we can get whichever semantic combination we desire.

In Wile's fielded POPART system, the desired result is to have Compaction be applied to the result of Post-Filtering, and to have both Compaction and Post-Filtering applied to the result of Additional Parsing. A serialization that will achieve this is to apply the evolutions in the following order: 1) Additional Parsing, 2) Post-Filtering, 3) Compaction, 4) Pre-Filtering, and 5) Rapid Rejection. The result of this is:

```
relation P(s,pt)
iff (not RR(s)) and PRE(s) and
    pt = COMP(POST(( IP(s,?)  ⊔ AP(s,?)  ),?),?)
```

Alternatively, should it be decided that neither post-filtering nor compaction should be done on the results of additional parsing, we achieve this by moving the additional parsing evolution to after post-filtering and compaction, and serially replay to get:

```
relation P(s,pt)
iff (not RR(s)) and PRE(s) and
    pt = ( COMP(POST(IP(s,?),?),?)  ⊔ AP(s,?)  )
```

We have used our transformation system to do the above evolutions replays. This is easy, since transformation invocations are recorded as objects, and can be ordered as desired and replayed on the original specification. To get the alternative results we simply reorder the transformations and mechanically replay them.

4. How Combining Evolution Transformations Facilitates Replay

Our thesis is that evolution transformations record how a specification has been changed, and that this knowledge is useful when it comes to replaying a transformational development on the original specification. We summarize the advantages of using evolution transformations.

The evolution transformation invocations are available for comparison, rather than only the end products of those transformations. For example, we compared the transformation invocations that introduced post-filtering and pre-filtering and predicted their independence. Likewise, we compared invocations that introduced post-filtering and compaction and predicted their dependence. This reasoning was done in terms of the transformation invocations themselves.

Alternatives during replay are apparent, and easily achieved. When a specification is changed, it may be ambiguous how to replay a development. Some kinds of ambiguity manifest themselves as the differing results that emerge from alternative orderings of replaying the transformations. For example, the choice of ordering of post-filtering and compaction leads to either compacting the results of post-filtering, or vice versa. Either of these is trivial to achieve, by simply replaying the evolution transformations in the appropriate order.

The generic nature of change is captured in the generic properties of evolution transformations. Our transformations were instances of a generic 'splice' operation. As such, we knew that interaction would only occur *if they were applied to the same* 'link' in the original specification, and that any such clash would give a choice of two alternative orderings of splicing in the two intermediaries into the link in question. This generic-level reasoning is easy to do, and, since it covers a broad range of cases, is worthwhile automating.

Evolution transformations motivate the choice of the appropriate form of specification. Realizing that our changes all concern the transfer of expressions across links encourages the adoption of a style of specification in which the links in question are explicitly represented. Hence the slightly modified initial specification in which `P` was defined as an explicit equality between the original parse of its input string, `s`, and its result, a parse tree `pt`, as in: `pt = IP(s,?)`. Conversely, realizing that none of our changes were concerned with a particular programming-language representation of failure (NIL returned to represent failure to parse) encouraged us to abstract from this into a neutral relation-oriented form in which parse failure is not a special value or aspect of control, but simply the absence of the relationship holding between the input string and any parse tree.

Replay's 'reference' problem is simplified in the framework of evolution trans-

formations. During serial replay of transformations, examination of the earlier ones can identify when later transformations' references to locations in the specification might be (inadvertently) affected. In the case of splice transformations, canonicalizing the form of those references and suitably ordering the transformations circumvents this problem. A more general approach could be based on having the evolution transformations maintain correspondencies between portions of the specification versions. We feel this latter approach has promise, but have yet to pursue it.

Layers of transformations remain useful, and are easily discerned. Wile's fielded system embodies one more change, namely the optimization of caching the intermediate results during parsing (sometimes called 'memoizing'). The parse function is defined recursively, and, in the course of parsing a lengthy string, many of its substrings may be parsed repeatedly if this function is implemented naively. Caching the results of those intermediate parses can give a big speedup, and is thus a valuable optimization [3, 14]. Note that this change, in contrast to the five that we considered, operates not on the interface between the parsing function and the rest of the system, but on the internal workings of the function itself. This is clearly apparent when considering the transformations to achieve those changes, and the level of specification upon which they operate. We would not try to combine these very different changes all at once, but would instead divide the transformation into distinct layers:
1) Pre-Filtering, Post-Filtering, Compaction, Rapid Rejection, Additional Parses
2) Represent failure as a value (NIL)
3) Memoize (cache) recursively defined functions.

The evolution viewpoint gives us a clear idea of what the layers should be. Also, the effects of one layer's changes can be understood with respect to the concerns of another layer. For example, the first layer introduces functions (`PRE` for pre-filtering, `POST`, etc.), which the third layer must consider as candidates for memoization.

5. Conclusions

We have argued that evolution transformations offer an alternative viewpoint of maintenance in the transformational development paradigm. This viewpoint suggests ways in which evolution transformations might increase and augment the repertoire of available techniques for making transformational developments more robust (replayable). A simple example, successfully run on our system, demonstrated how several changes to a portion of a fielded system could be concisely captured as transformation invocations, and this representation used to reveal the alternative possible combinations and realize the chosen one.

References

[1] R. Balzer, N. Goldman, and D.S. Wile. On the transformational implementation approach to programming. In *Proceedings, 2nd International Conference on Software Engineering, San Francisco, California*, pages 337–344, October 1976.

[2] F.L. Bauer. Programming as an evolutionary process. In *Proceedings, Second International Conference on Software Engineering, San Francisco, California*, pages 223–234. IEEE, 1976.

[3] R.S. Bird. Tabulation techniques for recursive programs. *Computing Surveys*, 12(4):403–417, December 1980.

[4] J.M. Boyle and M.N. Muralidharan. Program reusability through program transformation. *IEEE Transactions on Software Engineering*, SE-10(5):574–588, 1984.

[5] J. Darlington and M.S. Feather. A transformational approach to program modification. Technical Report 80/3, Department of Computing and Control, Imperial College, London, 1980.

[6] M.S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, January 1982.

[7] M.S. Feather. Constructing specifications by combining parallel elaborations. *IEEE Transactions on Software Engineering*, 15(2):198–208, February 1989. Available as research report # RS-88-216 from ISI, 4676 Admiralty Way, Marina del Rey, CA 90292.

[8] M.S. Feather. Detecting interference when merging specification evolutions. In *Proceedings, 5th International Workshop on Software Specification and Design, Pittsburgh, Pennsylvania, May*, pages 169–176. Computer Society Press of the IEEE, 1989.

[9] S. Fickas. Automating the transformational development of software. *IEEE Transactions on Software Engineering*, SE-11(11):1268–1277, November 1985.

[10] A. Goldberg. Reusing software developments. In *Proceedings, 4th Annual Knowledge-Based Software Assistant (KBSA) Conference*, 1989.

[11] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM TOPLAS*, 11(3):345–387, July 1989.

[12] W.L. Johnson and M.S. Feather. Building an evolution transformation library. In *Proceedings, 12th International Conference on Software Engineering, Nice, France*, pages 238–248. IEEE Computer Society Press, March 1990.

[13] J. Mostow. Why are design derivations hard to replay? In *Machine Learning: A Guide to Current Research*, pages 213–218. Kluwer, Hingham, MA, 1986. Revised and condensed version of paper in Proceedings of the 3rd International Machine Learning Workshop.

[14] J. Mostow and D. Cohen. Automating program speedup by deciding what to cache. In A. Joshi, editor, *Proceedings, 9th International Joint Conference on Artificial Intelligence, Los Angeles*, pages 165–172, August 1985.

[15] H. Partsch and R. Steinbruggen. Program transformation systems. *Computing Surveys*, 15(3):199–236, 1983.

[16] M. Sintzoff, M. Weber, Ph. de Groote, and J. Cazin. Definition 1.0 of the approximation deva.1 of a development language. Technical Report ToolUse.TD.deva10.DD88c, Esprit Project #510: ToolUse, 1988.

[17] D.R. Smith, G.B. Kotik, and S.J. Westfold. Research on knowledge-based software environments at kestrel institute. *IEEE Transactions on Software Engineering*, SE-11(11):1278–1295, 1985.

[18] D.S. Wile. Program developments: Formal explanations of implementations. In *New Paradigms for Software Development*, pages 239–248. IEEE Computer Society Press, 1986. Originally published in CACM 26, (11), 1983, 902-911.

REUSE BY DESIGN: DATA ABSTRACTION VS. THE "TOP-DOWN" MINDSET IN AN OBJECT-ORIENTED ENVIRONMENT

Jeffrey W. Grubbs

Robert F. Roggio

Department of Computer and Information Science
University of Mississippi
University, MS 38677
601-232-7396

ABSTRACT. Utilization of an object-oriented environment for application development is no guarantee that software produced therein will be designed for reusability. Designer/programmer methodological bias coupled with a misunderstanding of the object-oriented approach produces marginally effective abstractions with ill-defined or inadequate behaviors. These inappropriate abstractions tie software components to a specific application context and severely restrict the opportunities for reuse. This paper examines a limited Smalltalk application whose development demonstrates such effects. The qualitative and quantitative advantages realized through redesigning it for reusability are also discussed.

1.0 INTRODUCTION

Software reusability has been cited as a method of dramatically improving the quality of software produced, as well as reducing its overall associated development time and maintenance costs. Despite projections that through reuse overall productivity may be increased by 40%, and maintenance costs reduced by up to 90% the practice of reuse has seen limited application. [8] Among the reasons given for its restricted use is the availability of appropriate tools and methods to support creation, storage, retrieval, and recombination of flexible software elements. [5,8,11]

Object-oriented languages have been promoted as removing many of the inhibitors of software reuse through their facilities for producing adaptable, encapsulated, abstract software elements. [5] But as Tracz points out "While certain language features do facilitate the development of reusable software, the language, in itself, is not enough to solve the problem." [8] Indeed some languages (Smalltalk, Ada®) in themselves may initially inhibit reuse due to the large number of classes, or library of data types provided by them, as well as the organization, or lack thereof, of these classes/types. [6]

An additional obstacle to reuse is the inherent and inhibiting bias that the designer/programmer takes with him into the object-oriented environment. This bias, or mindset is the result of operating within a top-down structured methodology associated with most traditional implementation languages (C, FORTRAN, PASCAL) as well as with more recently developed languages (Ada®). This "top-down" mindset becomes quite apparent through the designer's/programmer's choices for abstraction within a given application context. Most initially opt for procedural abstractions almost exclusively . This choice does not allow them to take full advantage of the possibility of reuse within the object-oriented environment. [6]

This paper will focus on the relationship or causal chain between the mindset of the designer/programmer, his choices regarding types of abstraction, available language facilities, and intentional design for software reuse as demonstrated in a simple Smalltalk/V Mac® application. The original implementation of this product will be examined for evidence of the effects of a "top-down" mindset and its related choices for abstraction upon the resultant design. This will be followed by discussion of how redesign with reuse as a priority was effected and what benefits, qualitative or quantitative, such redesign produced.

2.0 REUSE AND OOPS

Software reusability as a goal of software development has found an ally in object-oriented programming (OOP) that claims much toward practical achievement of that goal. Object-oriented languages provide methods of data abstraction that can achieve separation of the view of how data structures are implemented from the behavior they provide to client programs. This separation allows the effects of changes to the representation of data to be localized--abrogating the need to make changes in all of the client programs. More importantly, and central to the issue of reusability, the ability to encapsulate data structures and their associated operations as abstract objects (in an object-oriented language) allows them to be decoupled from specific application contexts--making their reuse more likely.

2.1 THE MINDSET

Unfortunately, simply using an object-oriented language does not ensure that reuse will be incorporated into any given software application. As Tracz, [9], points out, a person's mindset greatly influences the approach used in implementing a particular software design--programmers who have spent their careers using a top-down structured approach to implementation are likely to continue to work at least initially with that mindset even if transplanted to an object-oriented environment. The consequences may be reuse at a minimal level--reuse of already existing language constructs without the necessary effort to produce additional reusable objects. This results in a software product whose components are application specific, difficult to modify without widespread propagation of change effects, less understandable, and unlikely to be reused to any great degree. Dan Halbert, [6], further highlights the misunderstandings involved with practical application of the object-oriented principles of abstraction and inheritance:

> . . . many programmers are accustomed to thinking in terms of procedural abstractions, emphasizing actions and processes, rather (than) data and state. As a result, when they first try object-oriented programming they may map the procedural abstractions they would have created directly onto object type definitions. They have other difficulties too, such as implementing behavior on the wrong objects, or creating type hierarchies that correspond poorly to levels of abstraction.

This "top-down" mindset produced exactly the kind of effects described during the original implementation of the natural language processing application, Designer (discussed later). The mismatch of primarily procedural abstractions with an object-oriented environment (vice data abstraction in an object-oriented environment) resulted in an application that failed to take full advantage of the possibility of reuse within Smalltalk.

2.2 TYPES OF ABSTRACTION

Traditional functional decomposition as a method of modularizing an application focuses on the operations to be performed. The tendency is to proceed from a locus of control, the main module, and distribute the processing functions to subordinate modules. This distribution of function provides some insulation from change effects due to procedural/algorithm changes, but does not necessarily provide similar protection from effects due to changes in data representation. This is true since it is not uncommon for the procedural abstractions derived from functional decomposition to communicate by passing complex data structures--with these structures, or other data items, being sent like bees from module to module. Meyer points out that almost one-fifth "of the cost of software maintenance stems from changes in data formats. This emphasizes the need to separate the programs from the physical structure of the objects they handle." [5] Such a separation implies a necessary encapsulation of the data structures involved--hiding their internal structure from other parts of the program that utilize them. This separation can be accomplished by changing the emphasis in the software architecture from procedural abstraction to data abstraction. Meyer once again provides succinct advice to effect this change in emphasis:

> Instead of building modules around operations . . . and distributing data structures between the resulting routines, . . . [Use] the most important data structures as the basis for modularization and attach . . . each routine to the data structure to which it applies most closely. [5]

Meyer is effectively describing what should be the modus operandi within an object-oriented language in order to take full advantage of such a language's capabilities. In the context of Smalltalk, these data structures and associated operations used as the basis for modularization are simply the Smalltalk object classes and methods. As Halbert pointed out earlier though, even with the recognition that data abstraction should be emphasized there may be confusion about what constitutes a viable object (class). Since the current discussion centers upon abstractions (objects, software elements) as they relate to software reusability, then object viability should be examined within that context.

2.3 ADAPTABILITY

Software reusability is predicated upon products whose elements present an adaptable level of abstraction. "Adaptable" here is a relative term implying that those attempting to implement future products would be inclined to adapt existing elements (abstractions) before attempting to create new ones. Several factors are critical to designation of software elements as adaptable:

(1) *understandability* : if one cannot understand an abstraction as it exists--how can one be expected to adapt it? This means that clear documentation of functionality and explicit interface specification are required.

(2) *modularity* : if an abstraction (the object and all operations applicable to it) is properly encapsulated then transplanting it to a new application context should prove straightforward. Otherwise, one is required to search for hidden dependencies.

(3) *genericity* : is this a useful abstraction? Will it make sense in another application context? Is the level of abstraction such that few or no modifications are necessary for reuse in another context?

Genericity as used here has a different flavor than that described by Meyer, [5], where it was applied to Ada® and Clu modules defined using generic parameters. However, philosophically both of these perspectives relate a higher level of abstraction to decreased development effort due to the reuse of the same object/module in different contexts.

Each of the above adaptability factors may be regarded from more than just the perspective of implementing isolated object classes. Each must also be considered upon creation of any hierarchy of these "primitive" classes if the emphasis upon "data abstraction" (the object-oriented approach) is to be complete. Thus the nature of inheritance among object classes also becomes an issue for consideration if reusability is the goal.

2.4 INHERITANCE

Inheritance allows a layered (hierarchical) approach to implementation of the details of an abstraction or a group of similar abstractions. This layered approach permits a user to deal with as many or as few of the implementation details of an abstraction as are necessary for a particular application. Additionally, the layered approach to inheritance can allow the commonality among a group of related abstractions to be "factored out" and placed in a higher level abstraction, or conversely allow extensions of a higher level abstraction to be expressed among similar, but unique, lower level abstractions. [4] Within Smalltalk this layered relationship among abstractions is expressed in terms of superclasses and subclasses. A superclass, ideally, expresses the common aspects among a group of lower level abstractions, its subclasses, or provides a view of a single subclass that is more dissociated from implementation details. It should be clear that either relationship can promote the reuse of the element(s) involved if appropriately structured. However, the adaptability factors discussed earlier apply here also. In addition to the considerations noted earlier for each factor there are additional considerations if the abstraction involved is a superclass:

(1) *understandability*: is the relationship between the superclass and each subclass clear--can one recognize it as a further abstraction of the properties of the subclass(es)? Do methods inherited by the subclass seem consistent with its relationship to the superclass and appropriate to its own abstract properties?

(2) *modularity*: are dependencies of the subclass(es) upon the superclass kept to a minimum consistent with their symbolic relationship, i.e., is access to the internal details of the superclass by the subclass kept to a minimum?

(3) *genericity*: is this a useful abstraction? Is there any greater degree of freedom or power gained by using this higher viewpoint (superclass)?

Considerations for the first point, understandability, indirectly address the problem pointed out earlier by Halbert--where is the best place in the hierarchy to implement a required behavior. Does one implement it within a subclass of an existing class, or create a new class?

The considerations for the second point, modularity, are particularly tricky--how much dependency is too much. Liskov states that this problem, which is really one of compromising encapsulation, is unavoidable in almost any system that uses an inheritance mechanism. "There are three ways that encapsulation can be violated: the subclass might access an instance variable of its superclass, call a private operation of its superclass, or refer directly to superclasses of its superclass. (This last violation is not possible in Smalltalk.)" [4]

Considerations for the third point, genericity, are almost inseparable from those for understandability--for if a superclass is useful in this sense, then its relationship to its subclasses should be clear. However, either a potential increase, or lack of change in design flexibility as a result of creating a superclass should not be overlooked when making a determination of usefulness.

Each of the adaptability factors will be addressed as it applies to both the original implementation of Designer, and the revised implementation (based upon design for reuse). Let us now examine the original implementation of Designer with the goal of identifying the effects of the "top-down" mindset in its construction.

3.0 AN APPLICATION

Designer was created as a demonstration of natural language processing within Smalltalk, and as a precursor, or front-end, to an automated object-oriented design tool. The underlying premise for the project is that object-oriented design starts with a natural language description of the strategy that is to be followed to solve the design problem. (Other steps in the object-oriented design methodology are well-documented, e.g., [7]) Designer is constructed to process that natural language description--identifying the objects, associated operations and attributes applicable to it. Objects within Designer are identified with nouns and noun phrases from the initial description. Operations that act upon the objects are identified with the verbs from the initial description. Attributes of objects are found by examining the adjectives associated with the nouns and noun phrases identified earlier. This entire process revolves around a determination of grammatical structure for each sentence of the input design description in order to correctly identify the roles of object, operation, and attribute in each case.

3.1 OOPS VICE "OOPS!"

Figure 1 shows the code for the initial (coordinating) method, openOn:, for processing within Designer. Two things should be apparent from close inspection of this code--first, there is no cohesive, persistent entity modeling the central element being processed--the sentence; second, the code is not exclusively Smalltalk.

```
openOn: aString
        "The receiver is an instance of Designer, and aString is a paragraph
        to be processed. aString is divided into sentences which are
        processed individually by the Prolog program Recognition. Each sentence
        is examined to determine what objects, operations, and attributes
        are within it. Objects are added to objectSet, operations to opsSet, and
        attributes to atribSet--each of which is transformed into a corresponding
        instance variable for the receiver--objects, operations, attributes."

    |acts aSentence atribSet descripStrings items num objectSet opsSet posit quals|

  Elapsed := Time millisecondsToRun: [
    descripStrings := aString formLists.          "descripStrings is an array of lists."
    objectSet := Set new.
    opsSet := Set new.
    atribSet := Set new.
    1 to: (descripStrings size) do:[:posit|
        aSentence := descripStrings at: posit.
        items := Recognition new :?               "Query the Prolog program Recognition"
                  objects(aSentence, p,q,r).      "for the objects in aSentence."
        objectSet addAll:(items unpackResult) asSet.
        acts := Recognition new :?
                  operations(aSentence,s,t).      "Query for the operations."
        opsSet addAll: (acts unpackResult) asSet.
        quals := Recognition new :?
                  attributes(aSentence,u,v,w).    "Query for the attributes."
        atribSet addAll: (quals unpackResult) asSet.
        ].
  ].

      "Instance variables to contain objects, operations, and attributes."
    desAtribs := Dictionary new.
    desObjects := Dictionary new.
    desOps := Dictionary new.

      "Sort and send objects, ops, and attributes as keys to the newly created dictionaries."
    (atribSet asSortedCollection) sendKeysTo: desAtribs dType: false.
    (objectSet asSortedCollection) sendKeysTo: desObjects dType: true.
    (opsSet asSortedCollection) sendKeysTo: desOps dType: false.

      "Open the application window."
    self createWindow
```

Fig. 1. Instance method openOn:

The absence of an object/class, other than temporary variables, modeling the sentences being processed is not bothersome from a functionality standpoint--the processing is effective regardless of its existence. However, within the context of laying the groundwork for future applications that incorporate natural language processing, the absence of such an object/class wastes an opportunity for reuse. Additionally, if changes were to be made to the sentence recognition method (to improve efficiency, or to facilitate semantic analysis) their effects could not be easily localized. Similar arguments could be made for the necessity of modeling the grammar (used for the recognition process) as an object/class.

The divergence from exclusive use of Smalltalk as the implementation language was precipitated by the characteristics of the language. As O'Shea, [6], and others have pointed out, the process of learning object-oriented languages, like Smalltalk, is complicated by a very large assortment (in some cases hundreds) of available classes/abstractions. In attempting to both learn the language facilities and meet the functional goals as stated for Designer within development time constraints a conflict was discovered. The conflict was resolved by using a Prolog compiler, available as another class in the Smalltalk hierarchy, to implement the sentence recognition algorithm. Preparation of the initial natural language description for input to the Prolog program, provisions for data structures to store the results, and windowing operations for display and manipulation of the resulting data were all implemented using standard Smalltalk/V® objects and methods.

The use of a combination of languages within Designer has arguable effects upon the reusability factors discussed earlier. Understandability may be seen as lessened by such a mix of approaches depending upon the expectations of future users, and their familiarity with both languages. Modularity is maintained from the perspective that the Prolog program models an object method. This "method" operates upon a representation of a sentence, and can be revised (within limits) without affecting portions of the application written in Smalltalk. However, if changes within the Prolog program (Fig.2) are considered from an interior view, then it is clear that data encapsulation is not enforced. The context-free grammar (CFG) rules, the schema for their application, and the related dictionary data are grouped together in such a way that changes in a single area may effect the entire program. This monolithic dependency-laden construction both directly and negatively impacts the third adaptability factor, genericity.

The Prolog "module" would need extensive modification in order to be useful in other approaches to natural language processing (i.e., other than top-down backtracking procedures without parse tree construction). Also, there are no methods available for changing either the CFG rules or the associated dictionary items (other than standard text editing). Finally, implementation of these (i.e., rule set and dictionary) as abstract objects would separate the data representations from the procedural aspects of the processing. This would allow simple substitution of various grammars and dictionaries while using the same processing algorithm, or use of the same grammar and dictionary with a different processing algorithm.

3.2 MINDSET EFFECTS

It should be clear from the two previous sections that the step-by-step implementation of procedural abstractions to transform input data into output data, the hallmark of top-down design, was the modus operandi in this application. Another indication of this approach may be seen in the

```
"Extract the attributes/adjectives of objects in a sentence."
attributes(w,atr,atr2,atr3) :-
    or((sen(w,[],_,_,_,_,atr,atr2,atr3)),sen2(w,[],_,_,_,_,_,atr,atr2,atr3)).

"Extract the objects from a sentence."
objects(w,obj,obj2,obj3) :-
    or((sen(w,[],obj,obj2,obj3,_,_,_,_)),sen2(w,[],obj,obj2,obj3,_,_,_,_,_)).

"Extract the operations/verbs from a sentence."
operations(w,ops,ops2) :-
    or((sen(w,[],_,_,_,ops,_,_,_)),sen2(w,[],_,_,_,ops,ops2,_,_,_)).

"Identify a sentence using grammar rules."
sen(x, y,obj,obj2,obj3,ops,atr,atr2,atr3) :-
    nph(x,z,obj,atr),vph(z,y,obj2,obj3,ops,atr2,atr3).

"Sentence construct 2"
sen2(x,y,obj,obj2,obj3,ops,ops2,atr,atr2,atr3) :-
    nph(x,z,obj,atr), cjvph(z,y,obj2,obj3,ops,ops2,atr2,atr3).

"Noun phrase"
nph(x,y,obj,atr) :-
    or(noun(x,y,obj),np2(x,y,obj,atr)).

"Secondary noun phrase"
np2(x,y,obj,atr) :-
    or((adj(x,z,atr),noun(z,y,obj)),(det(x,w),adj(w,z,atr),noun(z,y,obj))).

"Verb phrase"
vph(x,y,obj,obj2,ops,atr,atr2) :-
    or(vp1(x,y,obj,obj2,ops,atr,atr2),(verb(x,z,ops),pph(z,y,obj,atr))).

"Alternate verb phrase"
vp1(x,y,obj,obj2,ops,atr,atr2) :-
    or((verb(x,z,ops),nph(z,y,obj,atr)),(verb(x,z,ops),nph(z,w,obj,atr),pph(w,y,obj2,atr2))).

"Conjunctive Phrase"
cjvph(x,y,obj,obj2,ops,ops2,atr,atr2) :-
    or((verb(x,z,ops),cjn(z,w),vph(w,y,obj,obj2,ops2,atr,atr2)),
    (verb(x,z,ops),nph(z,w,obj,atr),cjn(w,t),nph(t,y,obj2,atr2))).

"Prepositional phrase"
pph(x, y,obj,atr) :-
    prep(x,z), nph(z,y,obj,atr).
```

```
"Adjectives"
adj(['document' |x],x,'document').
adj(['filed' |x],x,'filed').
adj(['index' |x],x,'index').
adj(['request' |x],x,'request').
adj(['retrieval' |x],x,'retrieval').
adj(['search' |x],x,'search').
adj(['several' |x],x,'several').
adj(['user' |x],x,'user').

"Conjunctions"
cjn(['and' |x],x).

"Determiners"
det(['a' |x],x).
det(['an' |x],x).
det(['any' |x],x).
det(['some' |x],x).
det(['the' |x],x).

"Nouns"
noun(['acknowledgement' |x],x,'acknowledgement').
noun(['document' |x],x,'document').
noun(['documents' |x],x,'documents').
noun(['examination' |x],x,'examination').
noun(['index' |x],x,'index').
noun(['indexes' |x],x,'indexes').
noun(['it' |x],x,'it').
noun(['location' |x],x,'location').
noun(['name' |x],x,'name').
noun(['parameters' |x],x,'parameters').
noun(['qualifier' |x],x,'qualifier').
noun(['request' |x],x,'request').
noun(['system' |x],x,'system').
noun(['terminal' |x],x,'terminal').

"Prepositions"
prep(['by' |x],x).
prep(['for' |x],x).
prep(['in' |x],x).
prep(['to' |x],x).
prep(['under' |x],x).
prep(['with' |x],x).

"Verbs"
verb(['activates' |x],x,'activates').
verb(['archives' |x],x,'archives').
verb(['destroys' |x],x,'destroys').
verb(['displays' |x],x,'displays').
verb(['files' |x],x,'files').
verb(['identifies' |x],x,'identifies').
verb(['maintains' |x],x,'maintains').
verb(['returns' |x],x,'returns').
verb(['retrieves' |x],x,'retrieves').
verb(['specifies' |x],x,'specifies').
```

Fig. 2. Prolog sentence recognition module

manner in which new methods were implemented. The formLists, countSentences, and unpackResult methods were implemented for the existing String, and Array classes respectively. They effected manipulation of many representations for what should have been a single representation of the central entity in the processing--a sentence. This multitude of representations for a sentence included a string, an array of substrings, a list, an ordered collection, and a stream object. Additionally, unpackResult, was necessary in order to transform the results from the Prolog recognition module into a format that the remainder of the application could understand. Instead of producing a coherent and complementary group of methods applied to a central, persistent entity--a sentence, the top-down mindset resulted in methods distributed among several competing temporary views of the same data.

4.0 REDESIGN FOR REUSE

Reusability was not included as an explicit goal in the original development of Designer. Redesign of the Designer application was undertaken with the goal of removing as many of the obstacles to reuse (as cited above) as possible. This included creating object classes modeling the central entity--the sentence, and the grammar used to recognize it. Other conditions placed upon redesign included: (1) implementing the application using the Smalltalk language exclusively, (2) consistent adherence to the object-oriented approach--focusing upon data abstraction, and (3) creation of abstractions that conform to the "adaptability" criteria discussed earlier--understandability, modularity, and genericity.

4.1 THE ABSTRACTIONS

Three new abstractions were created in order to meet the stated redesign criteria--Sentence, Grammar, and TDRecord. The Sentence and Grammar object classes were explicitly required by the redesign criteria while an additional object class, TDRecord, was found necessary in order to enhance conceptual simplicity in the sentence recognition process.

In choosing a representation for a sentence object class the ability to easily access individual words within a sentence was seen as the primary goal. Secondary goals included choosing a representation that would allow the sentence to recognize whether or not it had been successfully parsed, and to remember its grammatical structure once determined. A simple solution was found by modeling a sentence as an array of words--a subclass of Array with some additional instance variables.

Instance methods for sentence objects may be considered as belonging to two groups: those directly involved in the parsing process, and those which are applicable either before or after such a process takes place. In the first group topDownParse: effects the top-down backtracking recognition algorithm used to parse the sentence. It takes as an argument an instance of the object class Grammar--this allows substitution of various grammars while using the same recognition algorithm. The other method included in the first group is at:matches:using: which supplements topDownParse: during the parsing process. The second group of instance methods includes: (1) categories--which returns the contents of the *lexCategories* instance variable, (2) identify:addTo:--which is used to extract objects, operations, or attributes from a parsed sentence, (3) initialize: --

which initializes a new instance of Sentence with an array of words, and (4) isValid--which checks to see if the sentence has been successfully parsed.

The process of choosing a structure for a grammar abstraction was less constrained than that encountered for the sentence abstraction. The Grammar class is a subclass of the Object class, and it has two instance variables: *lexDictionary* and *rulesDictionary*. The instance variables are basically self-explanatory, *lexDictionary* contains a dictionary of words (as keys) with their corresponding lexical categories (as values) while *rulesDictionary* contains a dictionary of context-free grammar rules. Since Dictionary is an object subclass within Smalltalk there are existing methods which allow easy manipulation of both the word/category entries and the CFG rule entries. Instance methods for Grammar include: (1) findRules:--which returns all rules associated with a particular syntactic category, (2) lexicals--which accesses *lexDictionary* , (3) rules--which accesses *rulesDictionary*, and (4) with:with:--which initializes a new instance of Grammar with a CFG rules dictionary and a words/categories dictionary. This last method allows easy substitution of different sets of CFG rules and word/category dictionaries among instances of Grammar.

TDRecord models the record structure used in the sentence recognition process. The TDRecord class is a subclass of Array. Each instance of TDRecord provides storage for a single stage of the recognition process--indicating the state of the process at a particular position within the sentence. This approach emulates structures specified in the top-down backtracking recognition algorithm as given by Winograd, [10].

Instance methods for TDRecord are used primarily for reviewing or assignment of values in the fields of a particular record. This group of instance methods includes nextRule, posit, posit:, remainder, remainder:, rulesPending, and rulesPending:. These methods allow the previously mentioned review/assignment without concern for the underlying data structure. This hiding of structure is useful here since two of the record fields contain stacks, and not just simple data types.

4.2 REDESIGN OBJECTIVES VS. RESULTS

Figure 3 shows the code for the revised initial (coordinating) method, openOn2:, for processing within Designer. Upon comparison with Figure 1 it can be seen that there are no longer any calls made to the Prolog module, Recognition, so that the objective of exclusive use of Smalltalk for the implementation was met.

4.2.1 FOCUS ON DATA ABSTRACTION

Evidence that the redesign objective of focusing upon data abstraction was met can be found through further examination of openOn2:. Such examination reveals that there is still a transformation of the input design description from a single string to an array of "sentences". However, there is a difference between this transformation, executed by asSentences, and that performed by formLists in the original version of Designer. In the original version the transformation was undertaken to change input data into another format for further processing by the Prolog module. The data was transformed into lists that were passed to the Prolog module which did not return them. These lists constituted a temporary view of data created for the convenience of a procedural abstraction--they were not data abstractions. The asSentences method converts the input design description into an array of data abstractions--instances of the Sentence object class which exist (persist) before, during, and after the recognition process. Further, there is no need to transform results from the recognition process into a usable form. The topDownParse: method, unlike the Prolog recognition module, does not simply produce output data--instead it changes the state of each input sentence object. In examining the state of each of the parsed sentence objects, the identify:addTo: method determines the desired results without affecting that state.

```
openOn2: aString
   |aParagraph aSentence atribSet attributes objects objectSet operations opsSet |

 Elapsed := Time millisecondsToRun: [
   atribSet := Set new.
   objectSet := Set new.
   opsSet := Set new.
   attributes := 'ADJ'.
   objects := 'NOU'.
   operations := 'VER'.
   aParagraph := aString asSentences.
   1 to: (aParagraph size) do: [:i|
          aSentence := Sentence new: ((aParagraph at: i) size).
          aSentence
              initialize: (aParagraph at: i);
              topDownParse: OODGrammar;             "Attempt to recognize the sentence."
              identify: attributes addTo: atribSet;
              identify: objects addTo: objectSet;
              identify: operations addTo: opsSet.
        ].
 ].

    "Instance variables to contain objects, operations, and attributes."
  desAtribs := Dictionary new.
  desObjects := Dictionary new.
  desOps := Dictionary new.

    "Sort and send objects, ops, and attributes as keys to the newly created dictionaries."
  (atribSet asSortedCollection) sendKeysTo: desAtribs dType: false.
  (objectSet asSortedCollection) sendKeysTo: desObjects dType: true.
  (opsSet asSortedCollection) sendKeysTo: desOps dType: false.

    "Open the application window."
  self createWindow
```

Fig. 3. Instance method openOn2:

Another indication of the difference in focus between the revised and original versions of Designer may be seen in the conceptual simplicity of the code in the openOn2: method versus the confusion surrounding that associated with the openOn: method. More specifically, the code within openOn: (Fig.1) that references the Prolog module , as well as that associated with the subsequent transformation of the results, is confusing. It is cluttered with repetitive calls to the recognition module each of which is followed by immediate, and somewhat low-level, processing of the results of the call. Clearly this is a focus upon procedural abstraction. In contrast openOn2: (Fig. 3) shows a single application of the recognition method, topDownParse:, to the current sentence. This is followed by three successive applications of the identify:addTo: method to the current sentence (using a different argument in each case). However, unlike the corresponding sections of code in openOn:, the identify:addTo: method does not reveal lower-level manipulation of the underlying data structures. The result is a more succinct and abstract presentation of code that is easier to understand since it contains no unnecessary and confusing implementation details.

The single argument (OODGrammar) for the topDownParse: method also provides evidence of the focus upon data abstraction. Unlike the Prolog module wherein the dictionary of allowable words and the specification of the CFG rules were explicit--the structure of the dictionary and the exact nature of the CFG rules in the revised Designer are hidden. Consequently they (the rules and dictionary of words) may be included as a single argument to any recognition method, e.g., OODGrammar. Additionally, the behavior of this grammar class is described by its own instance methods. Again unnecessary implementation detail is hidden and encapsulated through data abstraction.

4.2.2 HOW ABSTRACT IS IT?

The final redesign objective addresses the need for adaptable abstractions. As discussed previously such adaptability is predicated upon the understandability, modularity, and genericity of the abstraction involved. The three data abstractions that resulted from the redesign of Designer were the Sentence, Grammar, and TDRecord object classes.

It may be argued that the Sentence class meets the understandability criterion in that its functionality is clearly documented through and its interface explicitly specified by the Sentence instance methods. Its behavior is understandable since the intuitive definition of a sentence as a collection of words remains unchanged by the class definition or instance methods. Modularity of the Sentence abstraction is demonstrated in that if one uses the class (definition and methods) in another application context there are no hidden dependencies upon other objects that would affect its basic behavior. The Sentence class demonstrates genericity since as an abstraction it would be useful within any application context that has a need for natural language processing. Additional behavior or functionality could be easily added via new instance variables and methods.

The Grammar class meets the adaptability criteria for reasons similar to those described for the Sentence class. The functionality of and interface specifications for the abstraction are defined by its instance methods. The class is self-contained, hence it can be moved to another application context without concern for hidden dependencies. It can be recognized as an abstraction associated with natural language processing (NLP) which makes it useful for other applications that utilize such processing. Modifications of Grammar for other NLP approaches, other than top-down backtracking, should be straightforward. For example, if one wanted to modify Grammar for use with an augmented transition network, a network abstraction could replace the dictionary of rules within the definition of Grammar.

The existence of the TDRecord class is somewhat problematic given a strict interpretation of the adaptability criteria. It is understandable as a kind of record, yet it is a very specialized record whose behavior (as specified by associated instance methods) makes sense only in the context of particular approaches to NLP. Although there are no "hidden" dependencies, an instance of TDRecord would demonstrate meaningful behavior only in the presence of a method like topDownParse: (from the Sentence class). That the TDRecord class demonstrates genericity is arguable also. In order for it to be used in other application contexts they would have to include the Sentence class. However, it is a useful abstraction in that it specializes characteristics of a standard superclass, Array. The TDRecord class also facilitates replacement of confusing low-level data structure manipulation syntax in the topDownParse: method with more understandable code.

The perceived advantage of creating the TDRecord class despite the inconsistencies with regard to adaptability criteria points out an area of potential conflict between the object-oriented approach and the goal of software reusability. Even from the limited perspective of the application under discussion it seems that such conflict is inevitable--there is always going to be a desire for specialization of abstractions even in designs dedicated to reuse.

4.3 SERENDIPITY IN DESIGNING FOR REUSE

Upon running the revised version of Designer it was discovered that the required execution time had decreased significantly. Some run-time measurements were taken for each version in an attempt to determine, quantitatively, the differences in performance between them. Given the same input, the average run-time for the original version (using openOn:) was 162.5 seconds, while the average for the revised version (using openOn2:) was 9.8 seconds. In round numbers that is almost a 17-to-1 improvement in performance. Upon further investigation it was determined that the three calls to the Prolog module accounted for approximately 93% of the block execution time for the original version.

It would be absurd to suggest that every effort at redesign for reuse will reap the reward of great increases in performance. In the absence of adequate documentation of the interface between the Prolog compiler and the Smalltalk system it cannot be determined whether the difference in performance lies in the interface itself or within the Prolog inference engine. Despite this lack of information one conclusion can be reached--if the redesign effort had not been undertaken the performance gain would not have been realized. The performance gain is a bonus for a redesign effort whose goal is not an immediate advantage nor specifically application performance-oriented. The goal of redesign for reuse is productivity-oriented and directed toward future development and maintenance efforts. Its immediate results are somewhat more subjective than run-time performance--these include increased understandability, modularity, and genericity of software elements/abstractions.

5.0 CONCLUSION

A survey of current literature on software reusability was undertaken subsequent to the original implementation of Designer. Upon reviewing the characteristics of the abstractions within that original version, and comparing them to the desired qualities in data abstractions intended for reuse--the authors were somewhat shocked to discover the existence of their own "top-down" mindsets. Despite exposure to object-oriented design principles, and an awareness of the desirability of data abstraction over procedural abstraction, one may yet be led to ignore these principles by such an unrecognized bias. When faced with unexpected development schedule slippages one may very well lapse into old habits--and for many these old habits are firmly entrenched in the top-down structured approach to design and implementation. The effects of such bias would most likely go unrecognized only in organizations in which software reuse and the object-oriented approach were relatively new goals. Yet these groups would be no less discouraged by the results of those effects.

Software reusability was nc: a goal in the development of the original version of Designer. However, had the developer's "top-down" mindset been recognized early on in the development process, then a measure of reusability would have been the natural result of applying the object-oriented approach. One may overcome the undesirable effects of the "top-down" mindset upon software reusability by: (1) clearly focusing upon data abstraction in the design effort, (2) recognizing that adaptability of abstractions determines to what extent they may be reused, (3) recognizing that operation within an object-oriented environment does not ensure that an object-oriented approach will be followed, and (4) realizing that exposure to object-oriented design and implementation principles does not guarantee immunity from effects of the "top-down" mindset.

Trademarks

Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).
Smalltalk/V is a registered trademark of Digitalk, Inc.

REFERENCES

[1] Bassett, P. G. Frame-based software engineering. *IEEE Software* 4, no. 4 (July 1987): 9-16.

[2] Biggerstaff, T., and C. Richter. Reusability framework, assessment, and directions. *IEEE Software* 4, no. 2 (March 1987): 41-49.

[3] Kaiser, G. E., and D. Garlan. Melding software systems from reusable building blocks. *IEEE Software* 4, no. 4 (July 1987): 17-24.

[4] Liskov, B. Data abstraction and hierarchy. In *OOPSLA '87 Addendum to the Proceedings*, 17-34, ACM, October 1987.

[5] Meyer, B. Reusability: the case for object-oriented design. *IEEE Software* 4, no. 2 (March 1987): 50-64.

[6] O'Shea, T., K. Beck, D. Halbert, and K. Schmucker. Panel: the learnability of object-oriented programming systems. In *OOPSLA'86 Conference Proceedings*, 502-504, ACM, September 1986.

[7] Pressman, R. *Software Engineering: A Practitioner's Approach.* 2d ed. New York: McGraw-Hill, 1987.

[8] Tracz, W. Software reuse myths. *ACM SIGSOFT Software Engineering Notes* 13, no. 1 (January 1988): 17-21.

[9] Tracz, W. Software reuse maxims. *ACM SIGSOFT Software Engineering Notes* 13, no. 4 (October 1988): 28-31.

[10] Winograd, T. *Language as a Cognitive Process, Vol.I: Syntax.* Reading: Addison-Wesley, 1983.

[11] Woodfield, S. N., D. W. Embley, and D. T. Scott. Can programmers reuse software? *IEEE Software* 4, no. 4 (July 1987): 52-59, .

SECTION 2

PANEL DESCRIPTIONS

ISSUES IN USING OBJECT ORIENTED DATABASES TO SUPPORT KBSA

Abstract for Panel at KBSA Conference

Subject: The Need for Multiple Levels of Granularity

Panelist: David A. Fisher

There is a need for multiple levels of granularity in program and project databases. More precisely, there is a need for efficient management of fine grain data. Without fine grain data, it is not possible to build semantic analysis tools, to achieve widespread interoperability, or to develop a component-based software economy. The relatively high cost of access and manipulation in large databases composed from small data items has traditionally precluded their use. There are, however, technical solutions involving multiple levels of granularity that can exploit locality to provide efficient processing of fine grain data.

Fine grain data is essential to all semantic processing and therefore to most useful software tools. Tools are cripplingly limited in their functionality and utility if they do not have access to the purpose and intent of the data they manipulate. This is true whether the data is policies, requirements, designs, or programs. It is impossible to ascribe meaning to data which is presented as opaque structures or only syntacticly. Semantic analyses require access to data at the substatement (i.e. individual word or node) level.

Inoperability is impossible when data is shared or accessed at the level of files or other larger objects with their internal structure unique to each processing tool. With efficient databases at the level of the nodes in the structure of sentences in programming, design, requirement, and policy specification languages, it becomes not only possible, but easy to build interoperable tools.

Interoperable tools communicating with fine grain data is a prerequisite to the small, reusable, often single function tool components that are ultimately required for a component-based software economy. It can be argued that the high cost of software and the lack of effective and widely available software tools is not for lack of knowledge nor limitations in current software technology, but rather our inability to reuse and exploit existing application and tool components. Instead, every component must be redesigned and reimplemented for each new application to produce the same expensive, large and monolithic applications over and over. Things have changed little from the early 1960's when Alan Perlis pointed out that in the software world we build on the toes instead of the shoulders of our predecessors. The solution is a software economy based on reuse of small application and tool components interoperating through the use of fine grain data.

Fine grain data can be processed efficiently by management of locality. Whether nodes, structures, or collections of related information, data at any level is not accessed randomly or in isolation. This locality of reference can be exploited by physically (not necessarily logically) structuring the data in multiple levels of granularity with larger objects composed from smaller ones. In natural languages, these larger structures are often referred to as context; in programming languages as modules, packages, or compilation units; and in projects may correspond to subsystems, programmers, or chronology.

Biography - David A. Fisher

David A. Fisher is President of Incremental Systems Corporation. His formal education is primarily from Carnegie Mellon University. He has been employed in industry, government, and academia; has published papers in the areas of compiler construction, distributed processing, bounded workspace algorithms, and code optimization; and has contributed to Ada from time to time.

He has recently been doing research in the areas of fine grain and semantically-based object management under the DARPA Languages Beyond Ada and Lisp project. He also completed a SBIR effort on persistent data for RADC last year. He developed the Iris internal form which is currently being used by the Arcadia Consortium and in the STARS effort.

OODBMS in KBSA

Aaron Larson
Honeywell Systems and Research Center
3660 Technology Drive
Minneapolis, Minnesota 55418

1 Persistent Object Store

At the request of the KBSA technology transfer consortium Honeywell did a survey of the KBSA developers asking what capabilities they needed/expected in an OODB. We only received one response, but based on it, our own understanding of the problem, and what we believe the other developers expect, there is likely to be some mismatch in the expectations of the developers and the capabilities of commercial OODBS. The primary issue is that of consistency constraints. The developers expect a fairly powerful, probably first order logic, constraint mechanism. Existing OODBS provide a much simpler constraint mechanism, usually a fixed set of predicates over the "slots" of stored objects. Optimizing very general constraints is quite difficult[1] and it is unlikely that commercial vendors are going to address this issue in the near future.

Another deficiency of existing OODBS is the general lack of a capability to store and use "methods" (i.e. code or behaviors) in the data base. Without this capability, the constraint mechanism is limited to only refer to the "slots", or representation, of the stored objects, rather than the method based abstractions on the slots. This is a clear violation of encapsulation, one of the primary benefits of object oriented programming. Furthermore, if methods are not part of the OODB, then maintaining consistency between the stored data and the programs that manipulate it becomes a configuration management problem. However, making the OODB store and run methods makes the OODB vendor create an execution environment sufficiently general to model the control primitives the users expect (e.g. should it have multi methods? multiple inheritance?, etc). Furthermore, optimization of method invocation has proven difficult when dynamic user specialization is permitted, plus transactions and configuration management issues are likely to make it even more difficult. And of course, for the foreseeable future the OODB will have to be accessible by programs written in different programming languages, otherwise existing software will be inaccessible. Making all this work in a distributed multi user environment is going to take a while.

These questions of course raise the issue of how tightly should the OODB be integrated with the languages and tools that will be manipulating it? If a tight integration is chosen, then the resulting system will be a huge monolithic environment, if loose integration is chosen then maintaining data base consistency will be very difficult.

[1]It is hard to determine the domain of the characteristic function for the set of objects which could change the validity of a constraint. Dynamically computing the dependents of a constraint based on execution traces is possible, but has a fairly high overhead.

2 Specifications of Object Oriented Systems

Writing specifications for object oriented systems is incredibly difficult. The primary cause of this is that object oriented systems do something that few other kinds of programming systems do; namely they "call back out". In other words, when a call is made to a "generic function"[2] it is very likely that it will call some other generic function which the user may have specialized (modified the behavior of). The problem is that if the circumstances under which the second generic function might be called are not very well specified, it is nearly impossible to specialize it correctly. This is primarily caused by the fact that specifying *when* a particular generic function will be called is exposing part of the underlying algorithm. Determining how much of the underlying algorithmic process to expose and what parts to hide is a difficult problem, essentially it is equivalent to predicting in advance what ways the system will be extended in the future. Specifying this requires a flexibility/optimizability tradeoff during the design of the system, something which has not typically been done rigorously in the past.

The last object oriented system we have written a specification for is the KUIE user interface toolkit. The textual description of KUIE is between 3 and 4 times the size of the source code and a (subjective) estimate is that only about 60% of the "ideal" specification has been captured. This fact alone makes one wonder if using natural language is an appropriate mechanism for specifying object oriented systems. Perhaps a better way would be to have the source code stand for itself either with some stylized commenting or perhaps by annotating the source to describe what part is the (hidden) implementation and what part is the visible specification. This issue is almost certainly going to arise in future stages of the KBSA program, the question to consider now is how much stylized notation is permissible in a "specification" document? Too much notation makes it difficult to get a good overall *understanding* of the problem, too little notation makes the specification too imprecise as a specification document.

3 Configuration Management

We believe that one of the major obstacles of software development the KBSA must address is representing information about the evolution of a system. Another is coordinating the activities of a development team. A unifying theme for these two problems is the ability to represent and reason about the change between acceptable states of a software system (i.e. changes between releases). Managing and reasoning about change will be a very large piece of the input to the policy engine of the KBSA activities coordinator. Good configuration management support is a high priority requirement for an OODB supporting the KBSA.

[2]A function whose behavior is described by a collection of methods.

DYNAMIC DOCUMENTATION AND EXPLANATION OF SOFTWARE

Panel: Dynamic Documentation and E· lanation of Software

Dr. William Swartout, Moderator
USC / Information Sciences Institute
Marina del Rey, CA 90292-6695
(213) 822-1511; swartout@isi.edu

A software system can be considered truly reliable only if it does what people expect it to do. This means that understandability of software is vitally important. Traditional software documentation is insufficient, because it easily becomes out of date, and because it does not present the information that people really need. What is needed instead is tools that can generate presentations and explanations of software automatically, based on the needs of individual users. This task requires expertise in a number of areas: programming psychology, user interfaces, automatic programming, and computer-based explanation and training.

This panel will bring together perspectives on the problem of automatic documentation and explanation and software. Questions that the panel will address will include the following:

- What kinds of information does documentation need to show? How does this depend upon the reader, and upon the task he or she is performing?
- Where does this information come from? Can it be extracted automatically from the program or its specification, or must it be supplied separately?
- What decisions must be made about how to present information? Who should make these decisions - the documentation tool, the user, or a human documentor?
- What media are most appropriate for documenting and explaining software?

The following researchers will participate in the panel.

- **Eduard Hovy**, from USC / ISI, is interested in natural language generation, text planning, and machine translation. His RADC-supported work has been in the area of generation of multisentential text.
- **W. Lewis Johnson**, from USC / ISI, is a principal investigator in RADC's KBSA program. His work and that of his colleagues has placed emphasis on mixing formal and informal descriptions of systems, and mapping between the two. He has also been active in the intelligent tutoring system area.
- **David Littman**, of George Mason University, is active in the areas of computer human interaction, intelligent tutoring systems, and the psychology of computer programming. He has studied the documentation problem extensively, and has developed techniques for generating more useful documentation.
- **Ursula Wolz**, from Columbia University, is interested in intelligent tutoring and explanation systems. Her recent work has focussed on providing relevant information to users of interactive environments based on the task at hand. She will address the question of whether individualization is truly possible, and describe some of the technology required to achieve it.

The panel will be moderated by Dr. **William Swartout** of USC / ISI, who is a leading expert in expert system explanation.

Partial Documentation is as Good as It's Going to Get!

Eduard Hovy
Information Science Institute of USC
Marina del Rey, CA 90292-6695
(213) 822-1511; hovy@isi.edu

In this discussion I will take the devil's advocate position in order to argue for realistic expectations and the idea of limited (or vague) documentation.

I will claim that it is impossible to produce useful documentation automatically for the simple reason that the system has to be given too much knowledge that is not directly reflected in the underlying data structures / code being used. Documentation is only useful if it successfully engages at the user's level of thinking and uses terms and concepts familiar or inferable. To reach this level of thinking, for most user's requirements, and to know how to structure the explanation and what to include or exclude, simply requires so much general semantic knowledge of the world and of typical users that it is beyond anything we can realistically enter into a computer today. Consider how difficult it is to understand even your own well-structured (of course) code written five years ago. Even knowing your naming conventions and programming style usually doesn't help; often you have to recreate a solution to the problem and then see how the code corresponds to it. How much more difficult is it then for a program, operating without general semantic knowledge of the world and without lexicons that tell it for example that `temp` and `hold` and `intermediate-variable` all mean roughly the same thing and are therefore likely to be used for the same purpose?

Take as example the entry for the Unix command `ls`:

```
NAME
     ls - list contents of directory

SYNOPSIS
     ls [ -acdfgilqrstu1ACLFR ] name ...

DESCRIPTION
     For each directory argument, ls lists the contents of the directory...

...more...
```

The actual Unix source for the `ls` command comprises a considerable amount of code. This code is not easy to make sense of when simply presented without preamble: it starts with a number of variable declarations, sets some switches, and after a large case statement which sets further switches, breaks into a number of seemingly independent modules which set further switches and flags. Even an expert programmer, given this code stripped of its comments and containing no overt clues (such as mnemonic variable names) to its intended functionality would find it hard to decode. Certainly, presented with all this information, coming up with the sentence "this code lists the contents of a directory" is beyond anything a program can be asked to do in the near future.

We all know this, of course. And we all know the answer: provide some background semantics. That's what KBSA is all about. We have even started classifying the types of background information required: user models, models of users' goals and tasks, models of the system and its components, knowledge of rhetoric and argument structure... all this and more is required. Many of these topics have been studied quite seriously in other fields: rhetoricians, the User Modelling community, discourse specialists. Yet to date no-one has succeeded in providing anything near an adequate user model, for example, that is capable of holding up over more than a few examples in a very constrained domain. The structure of discourse and the general principles underlying rhetoric are only gradually beginning to be understood. Work on the automatic construction of code from high-level specifications, probably the easiest of these endeavors, has produced no big successes in its twenty-year history.

Should we give up? Is this a hopeless task?

I don't think so. But I think we should be aware of our limitations and not aim for the impossible. I believe it is feasible to automatically construct descriptions of short blocks of code which achieve one primary, easily recognizable function. I do not believe it is feasible to construct descriptions of blocks of code of (in practise) longer than about 20 lines or so, or of code that serves two or more purposes at once. That is to say, our enterprise hinges on the representation and recognizability of functionality. And here we are lucky; though in many cases it may be impossible to infer or represent the full functionality of an entity or a block of code, it may suffice simply to capture part of it. In our daily lives we go a long way on partial functionality. If for example your car sputters and dies, and on opening the hood you see a cable dangling loose and the place it obviously plugs is gaping, you'd infer functionality of some electrical nature, plug in the cable, and drive on without a second thought, problem solved.

In the programming domain, partial functionality can very often be syntactically recognized or straightforwardly inferred. For example, in the LS code, the following case statement appears near the very top of the code and contains fflg:

```
while (argc > 0 && **argv == '-') { (*argv)++; while (**argv) switch (*(*argv)++) {
     case 'f': fflg++; break;
                    ...more... }
```

This is followed immediately by a conditional containing fflg:

```
if (fflg) { aflg++; lflg = 0; sflg = 0; tflg = 0; }
```

The same pattern is repeated for the other variables in the case. Without trying to understand what exactly is going on, it is clear that the case statement is selecting among various input parameter settings and then setting parameters that somehow control aspects of the output. A documentation statement that says the following:

This code sets output parameters depending on the following input parameters: fflg, ...

may not be startlingly deep but is correct and can be generated automatically. It may also be very useful to help focus the user's attention onto or away from this particular block of code.

I believe that no matter how powerful our automatic documentation producing systems, users are going to use them primarily to locate the areas of interest and are then going to perform old-fashioned human-like problem-solving/debugging. I argue that it is therefore not necessary to aim for complete and comprehensive automatic documentation production; it is more important to give a sense of the primary functionality, even in vague terms, than to spend too much effort on searching for the most appropriate level of detail and/or content.

Documentation that Clients Can Use

W. Lewis Johnson
USC / Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292-6695
(213) 822-1511; johnson@isi.edu

In conventional practice, there are basically two kinds of documents generated about systems. One kind of document is the set of reports generated in the early stages of the software development process, such as those mandated by DoD STD 2167A. These are typically written before coding begins. The other kind of document consists of user manuals, written after the coding is complete.

Documents generated in this fashion are likely to disagree with the functionality of the code. The KBSA program has been developing system description techniques which alleviate this problem. In the ARIES system, in particular, the system description developed during requirements analysis is used to generate both an executable specification for the system and a document describing the system. Diagrams of system structure can be generated as well, and simulation tools and domain-specific presentations can be used to visualize system behavior. Since the executable specification is the basis for deriving the implementation, correspondence between the generated descriptions and implemented code are maintained.

Such techniques are generally useful for helping people to understand systems and their behavior. However, they are aimed more at analysts and developers than other people with an interest in the system, such as clients and users. I will devote the remainder of my talk to presenting some ideas on how to generate system descriptions aimed at clients. The purpose of such descriptions is to help clients to evaluate a system to see if it meets their needs. Some of these ideas are being pursued within current KBSA work, and some are issues for future work.

The basic problem is how to describe a system from the client's perspective. Ordinary system descriptions such as 2167A requirements documents contain masses of detail which are irrelevant to client concerns, or which are difficult to relate to client concerns. Furthermore, different members of the client organization have different perspectives on the system, depending upon how they are expected to interact with the system. In general, this implies being able to describe a system from multiple points of view.

As an example, consider the problems of perspective posed by a system such as an air traffic control system. A specification document for a typical air traffic control system contains masses of information about such things as computer system components and peripherals, data interfaces, system startup procedures, performance characterics, tracking algorithms, and much more. Of this, very little is of interest to a controller. A controller needs to know how the system will contribute to his or her task of controlling aircraft. What functions does it provide to the controller? What is the division of responsibilities between controller and system? Can the controller reliably depend on the system to perform the functions it is expected to perform? Does the system perform those functions in a manner consistent with the controller's expectations, and with standard procedures and FAA regulations?

The first step is to give analysts the means to capture client concerns in the first place. One way to achieve this is to tie appropriate source documents to the system description. In many domains, including air traffic control, such documents already exist, such as FAA manuals describing air traffic control procedures. Hypertext facilities are used in ARIES to create links between fragments of such documents and the system description. Another important step is to explicitly model the activities of users such as air traffic controllers, their responsibilities, and the overall goals that they seek to achieve, such as orderly and expeditious flow of air traffic. A dependency trace between client goals and system specification components is maintained.

Next, client-understandable presentation media are necessary. Our initial investigations lead me to argue that a mixture of natural language generation and domain-specific graphical presentations should be used. For example, many air traffic control concepts can be most easily graphical using graphical depictions of controlled air spaces. Natural language captions embedded in the diagrams greatly increase their explanatory power. Domain-specific presentations are greatly superior to diagrams aimed at computer scientists, such as Petri nets and Statecharts. Furthermore, a documentation system should be able to present the same material using different presentations, in order to facilitate validation and minimize opportunities for misunderstanding. A number of researchers are currently investigating multimedia interfaces, but little of this work has yet made its way into the realm of requirements acquisition and validation. We have been investigating simple object-oriented architectures that facilitate rapid development of domain-specific presentations that can be readily integrated with textual media and generic graphic presentations.

Client-oriented documentation systems should be strongly oriented toward examples. Specific scenarios and situations are easier for clients to understand and validate, and provide a convenient way of suppressing tangential details that general system descriptions are so full of. We have been experimenting with a presentation architecture that supports strong interaction between example generation and presentation. When a user asks for a depiction of a concept, the system searches its simulation database for an illustrative example involving other concepts that the presentation system is capable of presenting.

Selecting what information to present, as others on this panel have argued, may require an understanding of the knowledge and goals of the user, of the documentation system itself. In the case of documentation, a key concern is whether the user is attempting to get a basic understanding of the system or is actively critiquing the specification. Capabilities developed both in explanation systems and in critiquing systems may be required in order to perform this task adequately.

Position paper by David Littman not available at this time

Extending User Expertise in Interactive Environments

Ursula Wolz
Columbia University, Department of Computer Science
New York, New York 10027
(212) 854 - 8124 email: wolz@cs.columbia.edu

Interactive environments often present an inherent conflict between ease of use and functional power and extensibility. Power breeds complexity, and learning to exploit that complexity can be a major impediment to efficient use of an environment. A critical component of this problem's solution is how users can effectively extend their expertise without suspending the task at hand. Our research at Columbia addresses the problem of providing personalized assistance to a user through an on-line question answering system called GENIE. GENIE generates textual information calibrated both to the task at hand and the user's past experience with the system. The work encompasses natural language generation, knowledge representation and planning for task-centered settings, and intelligent computer assisted instruction.

The perspective of this work is distinctive in that GENIE explicitly follows both linguistic and pedagogical principles, first by responding informatively and second by opportunistically attempting to enrich users' knowledge. Three kinds of expertise that exploit a three part user model are necessary to accomplish this. The expertise is characterized as follows:

- **Domain expertise** describes both the casual relationship between actions and their effects, but also describes the relationship between domain-specific goals and plans to satisfy them. In particular, it is important to include *semantic* information about the trade-offs between potential plans for a goal.
- **Analytic expertise** decides what information to include based on the user's current computational goal within the current situation, and on previous knowledge of goal satisfaction, rather than on simple spectra of user expertise and functional difficulty. It selects the content both for responding to the question and enriching the user's knowledge.
- **Explanatory expertise** structures the answer by choosing from a set of pedagogical strategies. In order to maintain principles of clarity and conciseness, it employs a "revision" phase that fully integrates the responsive and enriching information into a coherent text.

The user model includes:

- **A situational context** that provides information on the activity in which the user is currently engaged.
- **A discourse context** that provides details about the user's elocutionary goal in asking the question.
- **A functional domain model of the user's knowledge** that is an overlay of the domain expertise, but that may include knowledge absent from the domain expert, including faulty knowledge.

Our work extends natural language generation by unifying the structural advantage of ATN based schemata with the flexibility of classic text planning. It extends knowledge representation and planning in task-centered settings by clearly separating knowledge of *intentions*, that is how domain specific goals may be accomplished, and *causality* that is, what the results of actions are. Furthermore it provides a mechanism by which semantic distinctions between plans can be encoded. Finally it presents a unique approach to intelligent computer assisted instruction since it does not encode a curriculum, but bases an instructional episode on the task at hand.

The position that has resulted from this work is that automating documentation and support for interactive environments is extremely knowledge intensive. This is nothing new. However, our emphasis is different. While we believe that there is a need for better ways to automatically capture and represent knowledge of the environment itself, we see an equally important need to consider what kind of explanatory behavior is relevant, and what kind of user model is necessary. In particular, the user model should emphasize what the user is doing and what the user is asking, rather than beliefs about what the user knows, or what the user's general preferences are.

This view is a result of an important outcome of our work. We developed a notion of "useful variability," namely to what extent did varying the user model contribute to the range of texts GENIE could produce, and were those texts "individualized" in some sense. We expected that the three components of our user model, the discourse context, the situation and the beliefs about user knowledge would contribute equally to the variability in the texts GENIE produced. Instead, we found that the discourse and situation contributed more significantly than the beliefs about user knowledge. In other words, our results suggest that it is possible to achieve a high degree of personalization by concentrating on the specifics of the question being asked, and the situation the user is in, without requiring a rich model of beliefs about the user.

This further suggests that it will be sufficient for the design process to produce a robust model of the actions and artifacts available in a system, and the tasks that may be accomplished with them. Our work shows that both the situation and the nature of the question can be derived from these. While automatically producing such a model is a daunting task, it is not nearly as difficult as automatically deriving beliefs about a particular user.

KBSA/CASE TOOLS

KBSA and CASE Tools: two approches to software development

Gilles Lafue, Panel Chairman

Gilles Lafue has been transfered to Europe by his employer, Andersen Consulting. Due to this move he has been unable to meet the proceedings publication deadline.

Bob Balzer, ISI

Due in part to the above mentioned scenario and communications problems, Bob's position statement is not available in this proceedings. We believe that his position is well known and that he has been available to the public quite recently (AAAI90).

Glover Ferguson, Andersen Consulting

Due in part to the above mentioned scenario and communications problems, Glover's position statement is not available in this proceedings. Glover Ferguson has been a leading developer for Andersen Consulting's scientific CASE tool product. Foundation.

An Overview of the Software through Pictures CASE Environment

Anthony I. Wasserman

Interactive Development Environments, Inc.
595 Market Street, 10th floor
San Francisco, CA 94105

Software through Pictures® (StP) is an integrated multi-user CASE environment, designed to run on a heterogeneous network of workstations. StP takes advantage of graphical user interfaces, network-based file systems, and the X Window system to provide users with transparent access to tools across the network. The base environment of Software through Pictures is comprised of an integrated family of graphical editors, a Document Preparation System, version control, locking and access control, and object annotation, all sharing a central repository. The Software through Pictures graphical editors support both structured and object-oriented development methods, including Structured Analysis (DeMarco/Yourdon or Gane/Sarson), Entity-Relationship Modeling (Chen), Structured Design (Constantine/Yourdon), Real-time Requirements Specification (Hatley/ Pirbhai), and Object-Oriented Structured Design (OOSD) (Wasserman, *et al.*).

The object annotation mechanism allows a user-extensible set of properties and values, including free text, to be associated with every diagram and with every symbol within every diagram. This annotation mechanism supports a diverse set of uses, including data type definitions, process specifications, management information, program design language, requirements traceability, aliases, and formal methods for software development.

The Document Preparation System supports the creation of design documents and reports, generating these documents from templates that describe their structure and content. Text, tables, and graphics can be freely mixed. A template definition language allows users to modify existing templates (such as those mandated under the DoD 2167A standard) or to create new ones. The templates provide access to diagrams and to the various properties of the various symbols in the diagrams, as defined with the aforementioned annotations. Output can be in PostScript® or in the formats recognized by the FrameMaker or Interleaf technical publishing systems.

Other supporting tools provide generation of data declarations for C and other programming languages, code frames, and SQL data definitions for several relational DBMS's, including Oracle, DB2, Informix, INGRES, and Sybase.

All of the StP tools communicate via a multi-user, object-based repository, implemented with a multi-user relational DBMS. Extensive programs for checking design rules verify the completeness and consistency of diagrams for the supported methods.

Software through Pictures was designed to be easily customized and extended, as well as integrated with other software tools. The open architecture of Software through Pictures, termed Visible Connections™, provides the end user and the environment builder with many different facilities for integration and customization. All tool interfaces, file formats, annotation templates, and document templates, as well as the repository schema, are published and easily accessible.Furthermore, almost 400 environment variables provide additional control over numerous properties of the environment, such as fonts, tool options, text editors, and the machine on which specific programs (tools) should be executed. All user-visible messages, including the content of the main menu, reside in text files so that they can be modified.

The main menu ("desktop") and the family of graphical editors are the most visible pieces of the Software through Pictures environment. The editors use an object-oriented drawing framework, with multiple windows and pop-up menus, that allows users to navigate through a set of diagrams. For example, a user can push and pop through a hierarchy of dataflow diagrams or can go from a name reference on a structure chart to an entity-relationship editor where that name is (or can be) defined. Access to repository information is global, and a browser allows direct access both to object definitions and to the diagram in which an object is defined.

Each editor has its own set of symbols and connection rules. Users can (at their own risk!) modify the set of symbols, the connection rules, the menu contents, and the default symbol sizes. This information is stored in textual *rules* files. Each such file includes the set of symbols, the menu name for that symbol, the scale size, the adjacency rules for symbols and the types of connections that can be made between symbols. In the OOSD editor, for example, the connections between two classes could be a visibility (uses) relationship, an inclusion relationship, or an inheritance relationship.

Each diagram and each symbol in each editor has an annotation template. The definition of a template is structured text, in a form that allows users to modify and/or extend the template definition to collect the appropriate set of property values and textual descriptions.

Users can save diagrams, generate information for the repository, check their work, and produce documents at any time. The project repository holds all of the information associated with a system or project. The repository contains information on the names and definitions of all of the objects, along with project history information, references to the diagrams produced with the editors, and locks that support sharing of the diagrams among members of a software development project. Since the repository is just a set of relations, an enviroment builder can easily extend the definition. Thus, for example, one could define new attributes or relations to help with some of the project management aspects of system development. Also, users can write their own queries, programs, or document templates to retrieve additional information from the repository.

An Object Management Library (OML) is interposed between the tools and the repository. In that way, users can work with the set of objects and their properties without having to gain access to the schema of the underlying database. This level of access is strongly recommended, as it shields users from the low-level details of the data model, and provides a better programming interface. The OML is currently being extended to provide a more general set of object management services.

The repository is placed within a hierarchical project directory structure. For each project, there is one or more ``systems," each of which has a substructure to hold the files produced by the graphical editors and the project database. The project database may be shared among two or more systems in a project (or in separate projects, depending on access control) by linking one project database directory into several system substructures.

In summary, then, Software through Pictures is built on an extremely flexible architecture that supports ongoing customization and tool integration, allowing comprehensive network-based environments to be built for specific programming languages and application domains.

INTELLIGENT INTERFACES TO RICHLY FUNCTIONAL SYSTEMS: MAKING KBSA USABLE

KBSA-5 Conference Panel Description

Intelligent Interfaces to Richly Functional Systems: Making KBSA Usable

Panel Overview

William Sasso, Andersen Consulting (Panel Chair)

The end-product KBSA system will formalize activities and products of the software development process, in order to provide a rich set of development functionality. But to use KBSA successfully, the developer will need to integrate multiple layers of knowledge. This complex of knowledge will include models of the application domain (such as Air Traffic Control), the development techniques (such as object-oriented design, simulation, or finite differencing) and representations (such as Petri Nets or class hierarchy graphs) available, the relevant support provided by KBSA, and the target implementation environment (e.g., a network of Sun Sparcstations). In order for KBSA to support this synthesis of complex models, we must begin to discuss specific issues such as the following:

- In a shared initiative system such as KBSA, what guidelines help us determine when to assign initiative to the user and when to assign it to KBSA?
- How can we most appropriately present and organize the complete set of KBSA functionality? At what level(s) of granularity should that functionality be presented?
- What are the most effective techniques for interactive support of the incremental refinement of queries, commands, and software itself?
- What are the most appropriate representations for objects, operations, and interactions central to the software development process?
- In KBSA's repository, a richly linked network of software development objects, how can a sense of current place and desired direction be maintained?
- How can we effectively package group support functionality for large-scale software development (e.g., integration of parallel development paths)?

This panel will address these issues by bringing together a set of informed and opinionated software usability specialists from some of America's leading institutions of software research and development.

Panelist's Position Statement
Elliot Soloway, The University of Michigan

In the KBSA community, there is clear recognition that interface issues need serious attention. The real issue underlying this concern is recognition that KBSA has reached a major juncture in its maturation process: real programmers are poised to use KBSA-like environments in their day-to-day professional activities. Experience has shown, however, that non-technical issues can be the downfall of genuinely wonderful technology, so -- sooner rather than later -- we need to deal with the following questions:

- *Measuring productivity and quality of the software process:* The methodology underlying KBSA, as articulated by Green et al [6], wasn't directed at improving software along the established productivity dimensions used by traditional software engineering (e.g., lines of code produced hourly). Thus rather than measuring KBSA's impact in conventional, product-oriented units of productivity and quality, we need to explore new, process-oriented measures of the software development. We may find that these measures are not quantifiable, but rather consist of carefully articulated explanations. Frankly, I don't see professional programmers giving as warm a greeting to KBSA as accountants gave to computer-based spreadsheets. What counts as convincing evidence?

- *Transplanting the new functionality of the KBSA environments -- in theory and in practice:* From Jackson to Yourdon, from data flow to object-oriented, the field has learned (1) how hard it is to effect significant change in software professionals and (2) the importance of having computer-based tools to reinforce and reify new methodological concepts and prescriptions. For KBSA to have an impact, we need to better appreciate how cognitive, social, and organizational issues impact both the adoption of the KBSA perspective and the continued evolution of the basic KBSA concept.

In my talk, I will address the veracity of the these claims and explore possible mechanisms by which the KBSA community may proceed to deal with them now.

Panelist's Position Statement
Peter Selfridge, AT&T Bell Laboratories

Introduction: The subject of this panel is the interaction and potential synergy of two different technologies, both of which determine the functionality and usefulness of a KBSA system. A fancy user interface (a phrase that now almost implies high resolution screens, mouse pointing devices, and Macintosh-like icons, menus, buttons, and the like) can be critical to the usability of a system, while the underlying functionality of a system must exist to be taken advantage of. We have built three Software Information Systems (SISs), which are a kind of KBSA designed to aid in the understanding of a large body of software or a large formal specification. In doing so, we have explored two related issues:

- What is the purpose of a given SIS?

- What interface is most appropriate for that purpose and its enabling technology?

Software Information Systems: Understanding, maintaining, and changing a large body of existing software (or formal specification) is complicated by lack of documentation, paucity of human experts, and complexity of the domain and task itself. Common to the difficulties is the problem of "discovery," gaining an overall understanding of the software before beginning a specific task. A Software Information Systems (SIS) is a system designed to aid programmers in understanding a large system. We have concentrated on SIS oriented towards discovery and reuse. Our general strategy has been to employ a Knowledge Representation (KR) System to represent some amount of knowledge of the domain and the software, and to embed the KR in an interface that allows the user to "query" the knowledge to get answers to specific questions.

Our first system, LaSSIE [5], represented a *high-level model* of the call-processing software of the Definity 75/85 PBX (the primary domain of all our systems to date) at the level of objects and actions. The purpose of LaSSIE was to allow the user to construct queries (such as "what actions are the result of a button push by the attendant?"), examine the matching instances computed from the Knowledge Base (KB), and reformulate the query so as to "browse" the KB in a convenient and powerful fashion. The model was manually generated, written in the KANDOR language, and embedded in the ARGON interface. This interface allows query by reformulation and other features which match the underlying purpose of LaSSIE. For example, after formulating the query, the user is graphically presented with a list of the matching instances and the detailed representation of one specific instance. This specific instance can be changed by pointing to another instance in the list, and is used to narrow or widen the focus of the original query (retrieving fewer or more instances respectively). In addition, some queries can be stated in a restricted natural language, and the user also has access to a graphical representation of the KB.

The second SIS, MView [1], emphasized the *integrated display of graphical information* generated automatically from source code. While LaSSIE contained some source code information, querying that information was cumbersome. MView allows the direct visualization of several kinds of cross-reference information as graphs, and enables the user to browse and query these graphs using the mouse. For example, in LaSSIE one would formulate a query expression such as "retrieve all functions called by the function 'collect,'" while in MView one can accomplish this retrieval with a single button push. The graphical views in MView are extremely easy to filter (reducing the amount of information presented), browse (navigate through a graph larger than the display screen), and query (display the graph of instances which match a certain restricting query). MView was used for several significant discovery tasks, including the determination of function calls responsible for message traffic between separate processes.

CODE-BASE, our current prototype, attempts to address several new aspects of an SIS

[10]. First, it uses the Classic Knowledge Representation System [2] to represent generic knowledge of the C language, the UNIX operating system, and code-level conventions for domain-specific code. Second, it provides access to a database of cross-reference information produced by a code analysis system, CIA [4]. Third, CODE-BASE allows some queries to address the actual source files (to fetch files that contain certain code patterns, for example). Finally, CODE-BASE embodies the hypothesis that, once an interesting set of code objects have been retrieved in response to a query, it will be useful to invent a new category or concept in the KB schema and populate it with the matches. In this way, CODE-BASE allows the user to extend the KB schema "on the fly." This last capability requires the interface to allow the user to add new nodes to the KB and view the relationships between new and old knowledge. In addition, the user is able to define a new category as conveniently as constructing a query.

Discussion: In the process of developing these SISs, we have become more aware of each one's (often unstated) purpose and the differences between them. More interestingly, the purpose of an SIS affects the kind of user interface which is appropriate for the system. For example, one can imagine an SIS which contains all kinds of knowledge about a system and runs as a general-purpose software oracle. In this case, real-time response time may be less important than the expressiveness of the interface. At the other extreme is an on-line software browser intended for use at programmer workstations. Here the emphasis is on speed and usability, and more graphical techniques are probably called for. In our case, LaSSIE is closer to the first example, while MView is closer to the second. CODE-BASE emphasizes the ability to add new concepts to the KB, and its interface reflects this emphasis.

Panelist's Position Statement

Loren Terveen, MCC and University of Texas at Austin

For the last several years, I have been pursuing research within the Human Interface Tool Suite (HITS) research project in the MCC Human Interface Laboratory. HITS is an integrated set of tools intended to support the development of collaborative multimedia interfaces to high-functionality systems.

A *multimedia interface* supports human computer interaction via more than one communication medium. For example, it might support interaction through gestures, graphics, menus, natural language, sketches, touch, and video.

A *collaborative interface* exploits knowledge about tasks, applications, interfaces, and users in order to help the latter accomplish their tasks more effectively. The collaborative interface will interpret ambiguous inputs correctly in context, phrase outputs in ways appropriate to the user's situation, and provide advice on efficient ways to accomplish the user's goals. To act collaboratively, an interface must be integrated -- events and objects in one part of the interface must be accessible in the other parts so that tasks can be split across interface components as appropriate and still function with users in a collaborative and integrated fashion.

The HITS work is motivated by the belief that the advancement of interface design necessitates understanding how to build collaborative interfaces. Collaborative interfaces increase peoples' productivity in computer-supported tasks by allowing them to work closer to their conceptions of tasks, freeing them from irrelevant computer-oriented details. Advanced forms of collaborative assistance will increase access to computer-mediated applications by a heterogeneous set of users and provide interfaces that allow richer exploitation of the powerful computational platforms of the future. For interfaces to be cooperative and adaptive, they must have representations of the user's task, the language of interaction, the application, and the user. Thus, to a substantial degree, the interface to the high-functionality, knowledge-based system will itself be knowledge-based. The crucial role of knowledge follows from the design goals of HITS:

- *integrated multimedia interfaces:* To create an integrated multimedia interface, as opposed to an interface that happens to allow multiple forms of input and output, the various modalities in the interface must be able to communicate with each other. This will happen naturally if all the interface components share a single formalism in which all accessible objects are represented in a unified way.

- *collaborative interfaces:* To enable a collaborative interface, we must provide the relevant knowledge to support reasoning about user actions and to produce effective assistance for those actions.

- *interfaces to high-functionality, knowledge-based systems:* The most important use for these interfaces is to support application programs whose range of functionality is rich enough to make simple interfaces inadequate. Thus, we focus on high-functionality applications, including knowledge-based systems.

Given the importance of knowledge in HITS, tools for the effective entry and use of knowledge take on a critical importance. My work has focused on the development of the HITS Knowledge Editor (HKE), a collaborative knowledge editing tool. Different versions of HKE have been implemented for several different representation systems, but the bulk of the research has been carried out for the CYC knowledge base [7]. The CYC knowledge base currently consists of about 35,000 units (equivalent to frames), each consisting of a set of slots. CYC units average 13 slots per unit, and 2 or 3 values per slot, meaning that each unit bundles about 35 assertions. HKE is both a basic HITS tool and an example of a collaborative multimedia interface. Collaboration is required because of the complexity of the knowledge editing task.

Knowledge editing is different from simple data entry because the user must understand the structure and content of the knowledge base well enough to locate information in a timely manner and add or modify the information in harmony with existing representational conventions. Knowledge editing is composed of two major parts: browsing and entry. Browsing involves acquisition of a model of the relational structure of the knowledge base and landmarks from which important data elements can be found quickly. Entry consists of the management of a coordinated series of changes to the knowledge base, or the creation of a cluster of related data elements. Browsing and entry are closely interleaved during a typical knowledge editing session. Significant problems encountered during the editing of large knowledge bases are

- *Navigation and Orientation:* Working effectively in large data spaces means quickly locating important data elements without losing the thread of the current task [3, 8] and quickly assessing the importance of any given data element to the current task. Both these problems are confounded by the complexity and density of elements in the knowledge base.

- *Visualization:* The "proper" view of a piece of information is highly dependent on the user, the task, and the type of information being viewed.

- *Utilization:* Data evolves with use. It is seldom the case that users are either purely consuming or purely producing information. Instead, they produce new knowledge in the process of consuming existing knowledge, and vice versa. In some sense, the process of knowledge entry never ends while the knowledge is in use. This means pure browsing or entry tools are less effective than tools that support interleaved browsing and entry.

HKE allows users to represent knowledge at a high level using graphical sketches. Rather than forcing users to construct complete and detailed specifications of a knowledge domain, HKE uses its knowledge of common patterns and constraints in the knowledge base to (1) suggest related issues that the user should consider, (2) fill in assumptions necessary to make sense of the user's specification, and (3) *detect problems*. HKE communicates its recommendations to the user through the objects in the sketch, using techniques like shading and highlighting to direct his attention to objects that require more work. When the user selects one of these highlighted objects, HKE provides additional resources that help him respond to its recommendations.

Panelist's Position Statement
Michael David Williams, Intellicorp

The coming generation of development *environments to* knowledge based systems must confront three problems: scale, flexibility, and performance. Over the last two years, we have been developing interface technology to deal with these issues. While we have not solved these problems for all time, we have identified a variety of new techniques and technologies that respond to them head on.

Scale: The next generation of knowledge based systems will commonly manage thousands of classes, thousands of rules, and hundreds of thousands of instances. This has led us to what we call *"the test of 10,000."* All interface proposals must be challenged with the question "What if there were 10,000 objects (or slots, or values, or ...)?" The challenge is directed at both presentation issues (e.g., how would you put 10,000 nodes in a graph in a form usable by a human?) and performance issues (e.g., how long does it take to form and present a graph with 10,000 nodes?). Not every component will be expected to address this test; for example, we wouldn't expect to see 10,000 facet types on a slot. The argument that your component will only be dealing with thousands of items can be a simple one; the case that it will deal only with hundreds of items must be tighter; and the argument that it need only anticipate tens must be pretty cleanly articulated.

Further, we think of the test of 10,000 more as a direction and slogan rather than a precise criterion. There are few – perhaps no – interfaces that can hold up to the strict test. But asking the question and forcing the creation of interface tactics helps drive us toward interfaces that scale well in the face of these large and complex knowledge based systems.

There can be a variety of responses to the test of 10,000. In our recent work, some have included explicit set management tools, modularization of applications, high data density presentations, ellipsis, and explicit data hiding strategies (including multiple and adaptable views over sets of data elements).

Flexibility: The human interface design space is simply too large and too complex to design in advance. Successful designs must be constructed with adaptation and refinement in mind at the outset. To support such "late binding" of design elements we have adopted a variety of techniques:

- a high level tool kit to construct browsers and editors;

- explicit, editable, customizable styles associated with each display mechanism; and

- "Probes" – an explicit interface gap filler paradigm.

Performance (response time): While it is true that performance and functionality often trade off against each other, it is also true that *performance is functionality*. How any given human interface component's functionality is used is sensitive to how fast that functionality is provided. Imagine a text editor that takes minutes to present the finished appearance of the document (e.g., an "nroff-like" system). This single parameter can have dramatic effects on the use of any given editor.

High performance knowledge based object systems that can be used reflexively represent one critical recent technological breakthrough. These are knowledge based systems that can be used to build the development interfaces for knowledge based systems. They enable the use of tool kit strategies, direct manipulation editors based on model-world representations (e.g., Steamer, SimKit, etc.) and other techniques based on a rich representation of the underlying systems.

Development of an event management system is another enhancement that supports our interface tool kit. It serves as an impedance matcher between the human interface (with reaction requirements in tenths of seconds) and the dynamic object system (with reactions in microseconds). Until now this mismatch -- of five orders of magnitude – has made WYSIWYG interfaces above object systems highly prone to update problems or what we call the "dance of death" -- the dramatic slowing of the underlying object system due to repeated updates of the interface during state transitions.

These are solvable problems! They are not abstract issues with uncertain prospects for solution. Over the past two years, we have been building systems that confront them head on. Intriguingly, thanks to recent performance breakthroughs we have used knowledge processing technology, especially dynamic object systems with full representation capabilities, to solve these problems.

References

[1] Belanger, D.G., Brachman, R.J., Chen, Y.F., Devanbu, P.T., and P. G. Selfridge. "Towards a Software Information System." AT&T Technical Journal, March/April 1990, pp. 22-41.

[2] Borgida, A., Brachman, R.J., McGuinness, D.L., and L.A, Resnick. "CLASSIC: A Structural Data Model for Objects." Proceedings of the 1989 SIGMOD International Conference on the Management of Data. Portland, OR: 1989.

[3] Carroll, J.M., Singer, J.A., Bellamy, R.K.E., and S.R. Alpert. "A View Matcher for Learning Smalltalk." Proceedings of the 1990 ACM Conference on Human Factors in Computing Systems. ACM Press. Seattle, WA: 1990.

[4] Chen, Y.F., Nishimoto, M., and C.V. Ramamoorthy. "The C Information Abstraction System." IEEE Transactions on Software Engineering. March, 1990.

[5] Devanbu, P. Brachman, R.J., Selfridge, P.G., and B.W. Ballard. "LaSSIE: A Knowledge-Based Software Information System." Proceedings of the 12th International Conference on Software Engineering. Nice, France: 1990.

[6] Green, C. D. Luckham, R. Balzer, T. Cheatham, and C. Rich. Report on a Knowledge-Based Software Assistant. RADC-TR-83-195. August, 1983.

[7] Lenat, D.B., and Guha, R.V. Building Large Knowledge Based Systems. Addison-Wesley. Reading, MA: 1990.

[8] O'Shea, T., Beck, K., Halbert, D., and K. Schmucker. "Panel: The Learnability of Object-Oriented Programming Systems." OOPLSA-86 Conference Proceedings. ACM Press. NY: 1990.

[9] Selfridge, P.G., and R.J. Brachman. "Supporting a Knowledge-Based Software Information Systems with a Large Code Database." Position paper for the Knowledge Base Management System Workshop, AAAI-90. Boston, MA: 1990.

[10] Selfridge, P.G. "Integrating Code Knowledge with a Software Information System." to appear in the Proceedings of the 5th Annual Knowledge-Based Software Assistant Conference. Syracuse, NY: 1990.

BRIDGING THE GAP

KBSA_5 Conference Panel Description

Bridging the Gap

Mary Anne Overman, Moderator
National Security Agency (NSA)
Fort George G. Meade, MD 20755-6000

The panel will discuss how we insert technology into bureaucratic organizations and bring the state of government closer to state of practice available in the commercial world. State-of-the-art technology such as KBSA will not have a chance of implementation in the government if we can not facilitate bridging the gap.

Among the panel members will be:

- Bill Liles, CIA
- Dennis Smith, Software Engineering Institute (SEI)

SECTION 3

DEMONSTRATION DESCRIPTIONS

KBSA CONCEPT DEMONSTRATION MODEL

REFER TO PAPERS BY

MICHAEL DEBELLIS
WILLIAM C. SASSO & MICHAEL DEBELLIS

Penelope - An Ada Verification System
Odyssey Research Associates
Ithaca, NY

Odyssey Research Associates is engaged in a research and development project in Ada verification funded by RADC and DARPA as part of the STARS program. We have designed a specification language based on Larch (called Larch/Ada) for sequential Ada programs. We have built a prototype verification system, called Penelope, that aids programmers in the interactive and incremental development of formally verified Ada programs. Penelope will be a part of the STARS software engineering environment.

What is the Penelope System?

The Penelope system is designed to formally prove that an Ada program is correct. The programmer uses a specification language that was specifically designed for Ada to state conditions on program execution. For example, one can state conditions on entry to and exit (including exit by raising an Ada exception) from an Ada subprogram. Based on the programmer's input of specifications and Ada code, the system generates verification conditions (VCs), which are statements in first order logic. The proof of these statements implies that the program satisfies it specification.

Uses of the Penelope System

The system is used to provide greater assurance that critical software is correct. Formal methods provide a mathematical proof that a program is correct, and reduce the reliance on ad hoc testing procedures for quality assurance.

An important use of the system is to enable the programmer to prove security properties of Ada programs. Many security properties can be stated as invariants; that is, if a subprogram satisfies security properties on entry to the subprogram, then we want to prove that these properties are satisfied on exist from the subprogram. The Penelope specification language allows us to state these security properties. We have used the Penelope system to formally specify and verify Bell-LaPadula security invariants for a small set of subprograms from the interprocess communication code of the Army Secure Operating System (ASOS). (TRW is designing and implementing ASOS to meet the A1 evaluation criteria.) Odyssey's use of Penelope to verify security properties of the ASOS was funded by the US Army CECOM.

We have also responded to Carl Landwehr's challenge to the formal methods community to specify and verify an RS-232 repeater. Our solution was specified in Larch/Ada and verified using Penelope. Work on specifying and verifying Ada code from NASA's library is currently in progress.

Penelope User Interface

The environment of the Penelope system is an interactive editor. The system interactively performs syntax, static semantic checking and VC generation as the user inputs annotations and Ada code. An error detected in the syntax or static semantics, is immediately flagged and the user makes the necessary modifications while still in the editor. The VCs are updated accordingly and these VCs are proved using a simple proof editor and checker. More generally, when the programmer modifies any statement or assertion, only VCs that have been changed as a result of those modifications need to be reproved.

Formal Basis

We provide a denotational definition of a programming language semantics more regular than Ada's, and argue informally that - subject to certain hypotheses - observable properties provable of this regular semantics are true of Ada. We have developed a general technique for developing predicate transformers from denotational definitions and methods for proving their soundness. This theoretical work covers a large portion of sequential Ada, including exceptions, side effects and goto. Our specification language belongs to the family of larch interface languages. Larch is an algebraic specification language designed by J. Guttag of MIT and J. Horning of DEC. We have devised a general framework for formally defining the semantics of such languages.

Penelope System Implementation

We have implemented a subset of what has been formally defined. Currently our system can verify programs written in a subset of Ada which includes packages (without private types), global variables, user-defined Ada exceptions, overloading of operators and whose control constructs and data typing are equivalent to Pascal (without access types and I/O). Libraries are not currently supported. Our main tool for implementation is the Synthesizer Generator (SG). The system is implemented in 12,000 lines of the functional programming language, SSL, which is the language of the SG, and runs on a SUN-3/60. The theorem proving capability if currently supplied by a simple natural-deduction style proof editor and checker for first-order logic. We are integrating the SDVS simplifier from The Aerospace Corporation into the system to provide simplification of the verification conditions. The user currently has a limited capacity to assist the simplifier during intermediate stages of VC generation. In effect, such interventions "distribute" the proof of the VC's and thereby help to manage its complexity and also permits some reusability of proofs.

Comparison to other Systems

Penelope is one of the few verification systems targeted for Ada. Its uniqueness resides in the mathematical techniques developed as a foundation for the system (invisible to the user), and in its support for the programming style advocated by such leading theoreticians as Edsgar Dijkstra and David Gries-- the style of developing specifications, programs, and proofs in concert.

The Gries-Dijkstra style is supported because Penelope is interactive, instead of batch-oriented. In a typical batch-oriented environment the smallest unit about which the programmer can reason is a subprogram. The work cycle is: write the program, generate VCs, attempt a proof, fail, modify the program, resubmit, etc. With Penelope the programmer can reason about program fragments. The VCs associated with program fragments can be used to guide development of the rest of the program.

PowerTools

Neil Mc Coy
Iconix Software Engineering, Inc.
2800 28th Street, Suite 320
Santa Monica, CA 90405
(213) 458-0092

PowerTools has the ability to import and export FreeFlow CDIF (Computer-Aided Software Engineering Design Interchange Format) data. The vendor-independent CDIF interchange format is the basis for transferring semantic and graphic information between CASE tools of like representations. The ability to share share data gives end-users the freedom to choose tools that offer best price/performance and user interface, without being tied to a specific vendor.

Freeflow is a Macintosh-based CASE (Computer-Aided Software Engineering) tool used to create and control flow diagrams, minispecs, and a data dictionary in accordance with the DeMarco structured analysis method.

The interchange is accomplished through another PowerTools module, ASCII Bridge. The new version of ASCII Bridge fully supports import and export functions for the CDIF data interchange format. Users can create structured-analysis, real-time, and object-oriented design diagrams, dictionaries, and charts, and move them back and forth between applications.

KAPTUR: KNOWLEDGE ACQUISITION FOR PRESERVATION OF TRADEOFFS AND UNDERLYING RATIONALES

CTA INCORPORATED
Rockville, MD

KAPTUR is an environment that supports the evaluation of potentially reusable artifacts throughout the software development process. The goal of KAPTUR is to harness knowledge gained through successive projects in a given domain, in support of new efforts. Only by understanding the decisions that went into past development efforts can developers intelligently reuse existing artifacts.

The fundamental concept in KAPTUk is the *distinctive feature*, which is any feature of an artifact that differs from common or recommended practice, or that represents a significant development decision. KAPTUR employs hypertext techniques to link artifacts according to their similarities and differences, and to link the distinctive features of an artifact to the supporting rationales, associated tradeoffs, and issues underlying the decisions.

An initial prototype of KAPTUR was developed in 1989. We are currently at work on KAPTUR '90, which builds on the initial prototype, adds some functions deliberately omitted in the first phase, and corrects some deficiencies that we discovered through demonstrating the environment. This year we are also taking the first steps towards introducing KAPTUR into a production setting in support of NASA's development of ground system software for unmanned scientific missions.

In this demonstration we will show both the initial prototype, delivered to NASA in November, 1989, and some of the enhanced capabilities of KAPTUR '90. Using the initial prototype and an example from the satellite operations control center domain, we will show how artifacts, distinctive features, and underlying rationales, tradeoffs, and issues are linked into a hypertext network. We will then show the graphical user interface and faceted catalog capabilities that are key enhancements in KAPTUR '90.

ARIES

REFER TO PAPER BY

W. LEWIS JOHNSON & DAVID R. HARRIS

KIDS: A Semi-Automatic Program Development System

Douglas R. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304

We have been working to formalize and automate various sources of programming knowledge and to integrate them into a highly automated environment for developing formal specifications into correct and efficient programs. Our system, called KIDS (Kestrel Interactive Development System), provides tools for performing deductive inference, algorithm design, expression simplification, finite differencing, partial evaluation, data type refinement, and other transformations. The KIDS tools serve to raise the level of language from which the programmer can obtain correct and efficient executable code through the use of automated tools.

A user of KIDS develops a formal specification into a program by interactively applying a sequence of high-level transformations. During development, the user views a partially implemented specification annotated with input assumptions, invariants, and output conditions. A mouse is used to select a transformation from a command menu and to apply it to a subexpression of the specification. From the user's point of view the system allows the user to make high-level design decisions like, "design a divide-and-conquer algorithm for that specification" or "simplify that expression with respect to context".

The user typically goes through the following steps in using KIDS for program development.

1. *Develop a domain theory* – The user builds up a domain theory by defining appropriate types and functions. The user also must provide derived laws that allow high-level reasoning about the defined functions. Our experience has been that distributive and monotonicity laws provide most of the laws that are needed to support design and optimization. Recently we have added tools to support the automated derivation of distributive laws. KIDS has a library of theories arranged in a hierarchy with importation links.

2. *Create a specification* – The user enters a specification stated in terms of the underlying domain theory.

3. *Apply a design tactic* – The user selects an algorithm design tactic from a menu and applies it to a specification. Currently KIDS has tactics for simple problem reduction, divide-and-conquer, global search (binary search ,backtrack, branch-and-bound), and local search (hillclimbing).

4. *Apply optimizations* – The KIDS system allows the application of optimization techniques such as simplification, partial evaluation, finite differencing, and other transformations. The user selects an optimization method from a menu and applies it by pointing at a program expression. Each of the optimization methods are fully automatic and, with the exception of simplification (which is arbitrarily hard), take only a few seconds.

5. *Apply data type refinements* – The user can select implementations for the high-level data types in the program. Data type refinement rules carry out the details of constructing the implementation.

6. *Compile* – The resulting code is compiled to executable form.

Actually, the user is free to apply any subset of the KIDS operations in any order – the above sequence is typical of our experiments in algorithm design.

The demonstration will illustrate the use of KIDS on a problem arising from the design of radar and sonar signals with optimal ambiguity properties - the enumeration of Costas arrays. We will demonstrate tools for developing domain theories, design of a backtrack algorithm, simplification and partial evaluation optimizations, finite differencing, case analysis, and data type refinement.

KIDS is unique among systems of its kind for having been used to design, optimize, and refine dozens of programs. KIDS will likely be useful in the near-term as an algorithm designer's workbench. It currently works best in application domains which are well-understood and whose foundation is readily formalized. Applications areas have included scheduling, combinatorial design, sorting and searching, computational geometry, pattern matching, routing for VLSI, and linear programming. We believe that KIDS could be developed to the point that it becomes economical to use for routine programming.

KUIE: KBSA USER INTERFACE ENVIRONMENT

REFER TO PAPER BY

BOB SCHRAG

SOFTWARE LIFE CYCLE SUPPORT ENVIRONMENT (SLCSE)

REFER TO PAPER BY

DEBORAH A. CERINO & FRANK S. LAMONICA

MULTIVIEW

REFER TO PAPER BY

CHRIS MARLIN

AUTHOR INDEX